Persistent Programming and Object Oriented Databases

Malcolm Atkinson and Ronald Morrison
University of Glasgow University of St. Andrews

Abstract

Persistent Programming Languages are defined as those languages which allow **any** of their values to have lives of **any** duration. The first ten years of research into those languages are reviewed. The motivation for such languages has increased. There are significant technological developments pertinent to their implementation. To obtain their benefits requires a radical revision of the architecture of computer systems, and a major commitment to the paradigm. Their potential as a foundation for implementing operating systems and database systems is described, as well as their obvious use for writing long-lived and large scale applications. The paper concludes by examining some aspects of object orientation, to consider the solutions offered by the persistent paradigm.

1 Introduction

It is now ten years since a group in Scotland began the implementation of a persistent programming language. This paper reviews the progress in those ten years, and then suggests the next major step in persistent programming research. The review concludes that persistent programming languages have the capability of supporting applications programs and many components currently thought of as system components. The next step will demonstrate the value of persistent languages as a foundation for system software. It will also realise the target of 'seamless' computing which that research began to explore ten years ago, and provide efficient implementation of this class of languages. Such a seamless system is expected to yield very large productivity improvements for the implementers of large application systems.

The review reconsiders the major issues in persistent programming. In section 2 we offer a definition of persistent programming languages, and show that they are a subset of database programming languages. We are concerned that some people have used the word "persistent" in a weakened sense, where it is not available to all data types. In section 3 the original motives for building persistence are rediscussed, and we see that the range of applications for which they are appropriate has expanded. In section 4 a brief history of persistent programming is given, as a summary of progress and as access to the body of literature. The interaction between programming language research and database research is of continuing importance.

The recognition of persistence as an orthogonal property of languages (section 5) leads to a discussion as to why all languages do not have persistence or related languages with persistence. First the issue of comprehendability is addressed and we show that a persistent programming language is intrinsically simpler than the separate language and database system it replaces. In

sections 6 and 7 the fundamental issues concerning performance are considered. It is argued that if we can make operating systems perform adequately, then we can also build persistent languages of adequate performance (section 8). This leads to an analysis of the role of the operating system and a proposal to replace it by an appropriate persistent language, leading to a coherent and seamless context in which to build applications. This raises two challenges: to the language designers to present the necessary range of functions in a way that can be understood; and to the architects and engineers to implement these functions, automating all the necessary physical and logical mappings.

Given such an implementation of an appropriate language, we may go on to use it to implement an operating system (section 9), and various database management systems (section 10), so that invested effort and experience may be transferred to this context. It is argued that without such a radical change to system architectures the potential simplicity of persistent languages cannot be realised, and implementation will remain difficult, as the 'old' system components conflict with the new.

To illustrate the radically different approach of persistence, we consider object oriented database systems as an example of contemporary research and technology in section 11. We make a proposal for implementing them using a persistent language. We also propose alternative, possibly better, ways of achieving their functionality using persistent languages.

2 Persistent Programming Languages

Persistence is a property of data values which allows them to endure for an arbitrary time. For example, heap technology is introduced into programming languages, to extend the persistence of data from the activation period of a block to the execution time of a program. This is still not full persistence since there is an upper bound (the execution time) to the longevity of data. It is as important that brief lifetimes (transience) should be included in persistence otherwise a programmer has difficulty with intermediate results.

We identify three principles which direct the provision of persistence:

i) *persistence independence* the persistence of a data object is independent of how the program manipulates that data object, and conversely, a fragment of program is expressed independently of the persistence of the data it manipulates;

ii) *persistent data type orthoganality:* consistent with the general programming language design principle of data type completeness, all data objects, whatever their type, should be allowed the full range of persistence;

iii) *orthogonal persistence management:* the choice of how to provide and identify persistence is orthogonal to the choice of type system, computational model and control structures of the language.

Compliance with these principles is a requirement for a programming language to be recognised as a Persistent Programming Language. Note that persistence independence implies that the language may not require the programmer to explicitly request movement of values between long term and short term storage. Implementations of persistence must achieve a consistent semantics for data, irrespective of its duration, for example, sharing of mutable structures must be preserved. Similarly, implementations must ensure that the presence of persistence does not weaken type checking.

3 Motivation for persistent programming

The initial motivation arose from the difficulties of storing and restoring data structures arising in CAD/CAM research. (See start of figure 1) Contorted mappings were needed to store arrays and graphs represented as references and records onto database structures. These contorted mappings had a number of costs:

i) they introduced concepts extraneous to the computation, obscuring code and confusing programmers;

ii) they did not precisely preserve information and were not subject to type checking, consequently they were a source of errors;

iii) it was difficult to incrementally translate in both directions and consequently more data than necessary was loaded and unloaded per program run;

iv) there were large computational overheads performing the translations; and

v) concurrent use of the data was excluded.

These contorted mappings are symptoms of a philosophical error, which has its origins in the limitations of computer technology at that time. That error is to divorce arrangements for using a computer as a store (databases) from arrangements for using it as a symbol manipulator (programming languages). Any constructs which are useful for organising or representing information in a store are relevant to formation and performance of the computation model. Similarly, operations and constructs needed in the calculation are pertinent to operations on the store (extraction, selection, 'spring-cleaning', etc). Furthermore, we expect design criteria applicable in one context to be applicable in the other (e.g. in programming languages, large scale tasks result in a recognised need for constructs to provide modularity - these will be needed in databases for the same reasons).

The loss of adequate typing arises in three ways. The transformation from the types in the programming language, to the types in the storage form, and the inverse mapping, are not checked to establish that one is the exact inverse of the other. There are higher level types, not made explicit in the mapping or stored data, and so their conventions may be lost. (The present effort for standard data interchange formats, e.g. EDIF, STEP, etc., addresses the final problem, but does not address the first two.) Persistent languages deal with the first two sources of loss, and some instances of the final source. This final source is further addressed by data models implemented in persistent languages, but there always remains some structures that programmers may not make explicit.

Persistent programming languages were created to solve the problems enumerated above. They eliminate discontinuities in the computational model, and economise on design and implementation effort by utilising the same concepts and constructs throughout the total computational system. They attempt to give equal importance to the computer as a 'filing cabinet' and the computer as a 'symbol manipulator'. This may be contrasted with databases (including OODBs) which allow the former to dominate, and with functional programming which allows the latter to dominate. In any given application it is possible that data or algorithm will dominate, but it is inappropriate for a data centred or algorithm dominated view to be built into a language which is the foundation technology of implementation.

Recently, many more applications, such as office systems and artificial intelligence have come to need a persistent language. We contend, in this paper, that persistent programming languages are also suitable for implementing database systems and operating systems.

We challenge the contention that there are separate classes of application - "data intensive" and "computationally intensive". Application domains which appear *ab initio* "computationally intensive" sooner or later develop large volumes of data, human processes which need to be interrupted and resumed, and multiple project contexts which demand all the functionality of database support. Similarly, application domains which appear to be "data intensive" eventually prove to have complex data, exceptional cases, and sophisticated programming. We therefore contend that the present discrimination is an artifact of the currently available technology, and that eventually, a wide spectrum of applications will benefit from orthogonal persistence.

4 History of persistent programming

A recent survey [Atkinson & Buneman 87] presents an overview of the treatment of persistence in DBPLs, here we summarise the history of persistent programming languages and show the related DBPL landmarks in figure 1, the notes explaining figure 1 appear in figure 2.

Year	PPLs	Relational DBPLs	Other DBPLs
1974	Need recognised [1]	Pascal/R[2] & Aldat[3] under construction	
1977	Design Automation Data Requirements[4]	Pascal/R[2] paper	
1978	Need for persistence identified[5]		
1979	Attempts at persistent Pascal & persistent Algol 68 Nepal designed [6] S-algol[7] implemented	Astral[8], Rigel[9] Theseus[10] proposals	
1980	PS-algol version 1[11] implemented		Taxis [12] SDM [13]
1981	Building Persistent [14] Object managers	Plain definition [15] published	Daplex [16]
1982	Shrines[17] Transitive closure problem[18]		
1983	PS-algol version 2[19] Galileo [21] EFDM [23] RAQUEL [24] Persistent Ada proposed [26]	Modula/R[20] built Adarel [22] proposed	Adaplex [25]
1984	PS-algol version 3 [27] Amber[28]		
1985	Napier [29] design begins Appin 1 Workshop [30] CPS-algol [32]		Poly [26]
1986	PS-algol version 4 [33]	DBPL [34] RAPP [35]	
1987	Appin 2 workshop [36] Roscoff workshop [37] Methodologies developed [39]		Quest [38]

Year	PPLs	Relational DBPLs	Other DBPLs
1988	Napier version 1 implemented [40] DPS-algol [42]		Oberon [41] Modula-3 [43]
1989	Persistent Systems track HICSS [44] Newcastle NSW Workshop [45] Oregon Workshop [46]		

History of Persistent Programming and DBPL landmarks

figure 1

These notes are sometimes abbreviated to citation of relevant papers. Dates used are mostly those of papers on the work which obviously lags by up to 2 years behind the actual work.

1) [Atkinson 74 a,b, 75, 76] all these mappings to text, relations or Codasyl model proved unsatisfactory.
2) [Schmidt 77]
3) [Merrett 77]
4) [Atkinson & Wiseman 77]
5) [Atkinson 78]
6) [Atkinson *et al.* 82] This language proposal proved too complicated, it proposed: inheritance, block structure, explicit name spaces, nested transactions, concurrency, objects and orthogonal persistence.
7) [Cole & Morrison 82]
8) [Amble *et al.* 79]
9) [Rowe & Shoens 79]
10) [Shapiro 79]
11) [Atkinson *et al.* 81] Orthogonal persistence for all the existing types in PS-algol.
12) [Mylopoulos *et al.* 80] Primarily a design aid in its early form.
13) [Hammer & McLeod 81]
14) [Atkinson *et al.* 83a, 83b, Cockshott 83, 87, 88a, 89, Brown & Cockshott 85, Brown 87, 89] A succession of versions .
15) [Wasserman *et al.* 81]
16) [Shipman 81]
17) Implementation of a POMS: shadow paging via VAX VMS memory mapping by Paul McLellan & Ken Chisholm - unpublished.
18) The transitive closure problem was identified at a workshop in UEA [Atkinson *et al.* 84].
19) Added first class persistent procedures [Atkinson & Morrison 85a].
20) [Koch *et al.* 83]
21) [Albano *et al.* 83, 85].
22) [Horowitz & Kemper 83].
23) An experimental version of Daplex, built using PS-algol [Kulkarni 83, Kulkarni & Atkinson 84, 86].
24) An experiment with building relational databases and HCI using PS-algol [Hepp 83].
25) [Smith *et al.* 83]
26) [Hall 83]
27) Addition of rectangular image types and other facilities to permit HCI programming [Morrison et al. al. 86a, b].
28) [Cardelli 85].
29) An intended successor to PS-algol [Atkinson & Morrison 85b].
30) The first international workshop on Persistent Object Systems, held at Appin, Scotland [Atkinson *et al.* 85, 88b].
31) [Matthews 85]
32) First experiment with concurrent persistent languages [Krablin 85].
33) Added to PS-algol: events, exceptions, and the callable compiler [Philbrow *et al.* 88].
34) [Matthes & Schmidt 89]
36) Second international workshop on Persistent Object Systems (see note 30), [Carrick & Cooper 87].
37) 1st International Workshop on database database programming languages, Roscoff, Brittany, France [Bancilhon & Buneman 88].
38) [Cardelli 87]
39) Methodologies for organising persistent programs [Cooper *et al.* 87, Dearle *et al.* 87].
40) An implementation of Napier88 revised from the original (see note 28) [Dearle 88, Morrison *et al.* 89]
41) [Wirth 88]
42) A design and prototype implementation for a distributed and concurrent persistent language [Wai 88].
43) [Cardelli *et al.* 88]
44) Proceedings of the 22nd Hawaii International Conference on System Sciences.
45) Proceedings of the third international workshop on Persistent Object Systems, Newcastle, NSW, Australia, January 1989
46) International Workshop on DBPLs, Oregon, June 1989 (follows from Roscoff, see note 37).

Figures 1 & 2 are presented for two reasons:

i) for the new research student in PPLs or DBPLs, to use as a guide when reading into the subject; and,

ii) to show that there is a considerable body of research into persistent languages which already interacts strongly with the DBPL and general programming language research.

4.1 Persistent Programming: where database and programming language research interact.

As an example of this latter interaction consider the search for effective bulk data types in persistent languages. Buneman and Ohori [Buneman 85, Buneman & Ohori 87, Ohori 87] have explored the integration of relation types with inheritance and record types, using a semantics similar to that developed by Cardelli [Cardelli 84] for multiple inheritance, and first exhibited in Amber [Cardelli 85]. This work by Ohori and Buneman was initiated in the early design discussions for Napier, as the relational type proposed for Napier generated a complex interaction of types [Atkinson & Morrison 85b]. It has led to a proposal for a language, Machiavelli, with extensional polymorphism [Buneman & Ohori 89] which they claim exhibits all the properties of object oriented systems, and is superior to Amber in avoiding loss of type information, when an extensional polymorphic procedure is used. Similarly work on persistence for functional languages [Argo *et al* 87] is the basis for potentially large scale data structures with optimised access [Trinder & Wadler 88, Trinder 89]. That method of organising bulk data derives from notations present in Miranda [Turner 87] and Orwell [Wadler 85] and has similarities with FQL [Buneman *et al.* 82a]. Other approaches to bulk data potentially include facilities for the programmer to define the appropriate type, if sufficiently rich type systems can be defined [Cardelli & Mitchell 88]. In object oriented systems the extent of classes are often the only bulk type. In O_2 [Bancilhon *et al.* 88, Lecluse *et al.* 88] there is an explicit set construct, as well as these extents. Leibnitz [Keedy & Rosenberg 89] provides both sets, and sequences with various forms of ordering. There are difficulties in arranging to optimise expressions involving these bulk types, in the context of languages which have objects or are data type complete [Zdonik 89]. The elaboration of this example is not meant as a survey of current work on bulk objects [Atkinson *et al.* 89a] in PPLs, but rather to illustrate the following aspects of persistent programming languages (and to some extent DBPLs) consequent on persistence being an orthogonal property of data:

i) that it benefits from research into programming languages;

ii) that, potentially, once persistence is a well developed concept, with good supporting implementation methods, it can be composed with any (nearly any?) good programming language design to yield a persistent programming language; and

iii) that it is the concern of PPL designers to face both the issues of programming languages and of databases and to synthesise designs that effectively address both domains.

These last two aspects are now considered further.

5 Persistence as a separate dimension

'Dimension' is used here, to indicate a property that can vary independently of the other properties of a programming language. We argue that the investigation of persistence is independent of the investigation of other aspects of the language, such as computational model, control structures, type systems etc.

But, if this is the case, why aren't there a plethora of persistent programming languages to match all the non-persistent ones?

To answer this we first consider the questions:

i) is persistence absent because persistence combined with an arbitrary programming language is intrinsically difficult to understand? and,

ii) is persistence absent because it is intrinsically too difficult to implement?

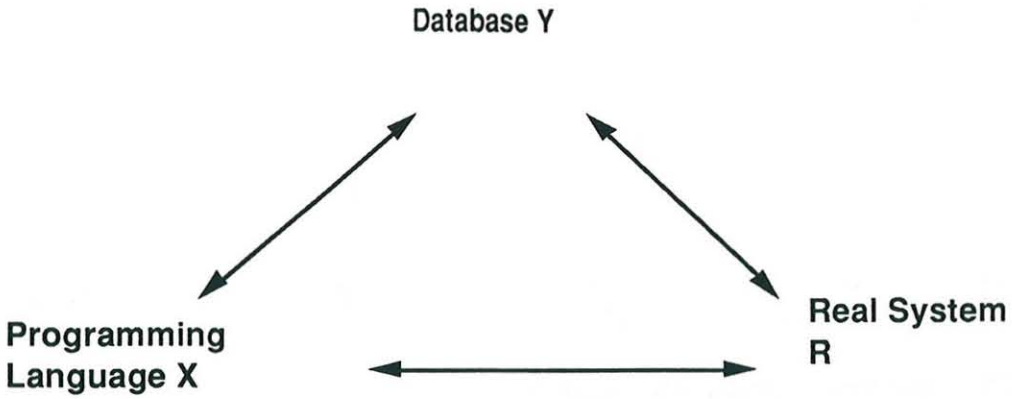
The former question is the more fundamental, since the understandability of a language (primarily for the programmers who use it, but also for those who implement it) is the most important property of any language. This question is a question about the nature of those programmers. To build a particular application they either have to:

a) understand language X and database (filing system) Y and the interface XY between them, or

b) understand language X', where X' is X with persistence added.

We contend that the latter option is intrinsically easier *for them.* If the data representation and operations of X and Y differ (if they don't the system reduces to X') then the representation of the same information will differ in X and Y. The programmer then has to organise the translations and movements of data between X and Y. In the case of X' neither these explicit translations nor the explicit organisation of movement are necessary. At present, in both systems, the programmer still has to assist with the organisation of concurrency, transactions, recovery, etc. In both systems problems of scale, distribution, name organisation, etc. may also arise. It is unlikely that separation of the support system into two components will help with any of these additional requirements, indeed, in general, such separation means that each has to be considered twice when using X and Y, but only once when using X'. (Even when using X' they may still be intrinsically difficult factors to specify and implement.) In reality, much of the present implementation of these factors in present day application programming depends on the use of a third support component, an operating system Z, which we discuss shortly (sections 8 & 9).

The two options may be summarised by the following diagrams:



USING A COMPOSITION OF LANGUAGE AND DATABASE

USING A PERSISTENT PROGRAMMING LANGUAGE

These diagrams emphasise the simplification achieved by PPLs. In the former, the applications system builder (attempting to model, administer or control a real system R) is concerned with maintaining three mappings: XY, YR and XR. In the latter, the applications systems builder (undertaking the same task) has to maintain correctly only one mapping X'R. This should be intellectually easier. Not only is the number of mappings reduced, but the possibility of inconsistency errors, where XY followed by YR is a different mapping (for some information) from the direct mapping XR, is eliminated. In general, the mappings have to operate in both directions. A consequence, in the former system, is that mapping X to Y followed by Y to X may not be information preserving. The avoidance of translation and, potentially, the support of type checking throughout the data's lifetime, eliminates this class of errors from the persistent programming system.

Philosophically, we can argue that if there was a case for two support components X and Y, they would evolve to be similar. Both are required to support models of the same set of real systems {R}. Eventually, any feature or concept which assists in building the mapping XR, will prove useful in YR and vice versa. Consequently they will both eventually be based on the same concepts, and there would be no *logical* benefit in keeping them separate.

The use of X' is neutral about the precedence of data and program. In contrast most combinations of X & Y give a data centred view. Design and decisions regarding the data precede the work on programs, and it is often difficult for the programmer to influence the model created in database Y as a result of insight developed while programming the application. In other cases, where Y is a filing system (which carries very little semantics about the data it stores) the programming decisions dominate. In a persistent system X', program and data have equal precedence and may be designed *incrementally, in either order.* Practice, disciplines, and methodologies may then choose any pattern of design, specification and construction that is appropriate to the application, without constraint from the implementation technology, X'.

These arguments imply that the addition of persistence to a programming language leads to an intrinsically simpler system to understand, for building a *complete application system.*

We therefore consider whether the difficulty of their construction and support is an impediment to their widespread use.

6 The cost of persistent systems

The question, "Is persistence absent because persistence is too hard to implement?" is interpreted here as a question regarding the cost of engineering to support a persistent programming language. We can review the support of a PPL as requiring three components:

A) a mechanism for translating the constructs in the language into appropriate data structures, including the representation of procedures (e.g. code generation);

B) a mechanism for interpreting (called 'executing', 'evaluating 'etc) those data structures; and

C) a mechanism for managing (creating, storing over a lifetime, etc) those data structures.

Mechanisms A and B for PPLs are not intrinsically different from the same mechanisms for other languages, see for example [Dearle 88]. Mechanism C, however, is the focus of much attention, and with the present state of widely available technology raises difficulties. It is, therefore, discussed in more detail.

Mechanism C can be divided into three subcomponents:

C_1 The provision of sequences of bytes of stable storage in which to store the values that represent the information;

C_2 The provision of addressing mechanisms for identifying the sequences of bytes of storage; and,

C_3 The provision of stores and interfaces which make those byte sequences and addresses consistently available to mechanisms A and B.

When stated in this form, these may be recognised as components of a typical operating system. For example, the segments of Multics [Organick 72] provide such sequences of bytes, and are addressed by segment numbers, and made accessible through an address faulting and paging mechanism. Why then isn't C trivially provided by copying the operating system technology? The reasons commonly put forward are two fold:

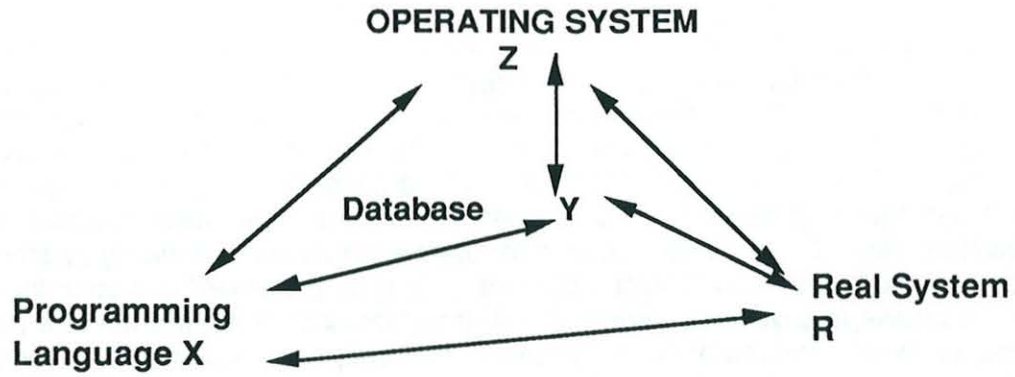
i) The populations of byte sequences have different properties from those in operating systems; and

ii) the stability requirements are more severe.

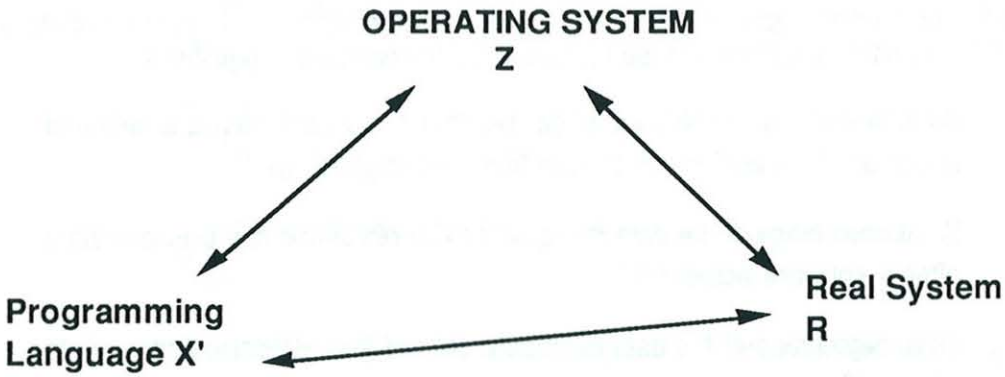
These differences may arise because the operating system offers the programmer facilities to manage physical mappings whereas, the PPL presents logical mappings. The differences are considered in turn. We will call the byte sequences 'chunks' [Atkinson *et al.* 83b], though they are

8 Operating Systems and Persistent Programming Languages

If the diagrams in section 5 are redrawn to show the operating system Z we get the following:



Using a composition of language, database and operating system



Using a persistent programming language and operating system

The relationships between Z and the other components are not all data and representation mappings as in the earlier diagrams. Z provides functions which enable X, Y & X' to operate, e.g. the ability to execute a machine instruction, to do a disc transfer, or to wait 5 seconds etc. There is a mapping of data across ZY and ZX', as the operating system may make (usually minor) changes to the bytes (e.g. adding framing data) when storing data on behalf of Y and X'. The interface ZR is typically active if R includes people, who then stimulate and communicate with the other components via the operating system.

Again we note that the additional support component introduces complexity. The applications programmer has to understand and use the operating system, while understanding and using the other components X &Y or X'. The interaction between Z and these components may not be easy to understand. For example, if some data is stored directly in Z's files, and other data stored via Y (or X'), then if the rollback facilities of Y (or X') are used to restore an earlier state, the programmer or user will be responsible for explicitly restoring those files to the corresponding state.

Again we therefore propose a simpler system:

USING A COMPLETE PERSISTENT PROGRAMMING LANGUAGE

In this system the functions of the operating system have been subsumed into the new persistent programming language X". The advantage is a seamless world for the applications programmer, where there is one coherent model of computation, not just calculation, covering everything necessary for application programming, in a *single consistent framework.* The conceptual advantage to the application's programmer is obvious, and there are many more application programmers than systems programmers.

There is a philosophical argument as to why it is likely to be both desirable and feasible. An operating system provides an abstract (higher level) machine,

F_1) Independent of the supporting hardware, e.g. that machine has process creation operations, file operations, character stream I/O operations, etc; Some of the principal functions of this abstract machine are:

F_2) To organise the concurrent execution of processes;

F_3) To provide and manage storage;

F_4) To provide a filing system, including stable storage, naming of long term data, protection, security, and incremental update;

F_5) To provide incremental delayed binding mechanisms, e.g. to bind a program, identified by a file name, to a process, then to bind data, identified by some other file names, to that executing process; and

F_6) To provide a control language enabling users to organise their computations.

But it is also the task of a programming language to provide an abstract (higher level) machine, with a semantics independent of the supporting hardware. If it were persistent then that programming language would also need to provide functions F_2 to F_5 consistently defined in F_1. If that language, as is now likely, also had an interactive (immediate execution) mode of operation, then it could also service F_6, otherwise an interpreter written in the language would service F_6.

Therefore, there is very considerable overlap in the functions of an operating system and a persistent programming language. Furthermore, they are trying to support the same people, manipulating similar data, with similar algorithms, tackling similar applications. Therefore, we would expect the requirements, loads, data properties, etc to be the same. The major difference being the expectation of seamlessness for the persistent programming approach.

If the operating system and the persistent programming language implement the same functions for loads described by the same parameters it is redundant effort to implement both, and it generates unnecessary complexity for programmers. Often the two implementations will conflict and interfere deleteriously.

The above argument is only sustainable if we accept a closed world hypothesis, i.e. that entire systems are implemented in *isolation* within this persistent system. Elsewhere we discuss ways of relaxing that hypothesis [Atkinson 89].

9 Persistent systems implement operating systems

Section 8 established that there is a large overlap between the functions provided by an operating system and a PPL. The work of implementing the run-time system of a PPL (we call this run-time system a "persistent system") may be potentially redundant duplication of the effort in writing an operating system. When they are implemented separately they will conflict (e.g. for physical memory space) and present a more complex support system for applications programming.

How should we proceed to implement X" to avoid such duplication? We conclude that the persistent system should be built directly on the hardware of the host machine. The operating system should then be built by writing code in X". This does not differ fundamentally from the approach which justified the design of Oberon [Wirth 88]. It differs significantly in detail, as it is intended that the functions of X" are much higher level, for example, they abstract over the store hardware, so physical store mapping is no longer a consideration for the operating system writer. Similarly processes and concurrency may be supported in the PPL [Morrison *et al.* 89b], so that any residual operating system functions are likely to be relatively trivial to implement. We illustrated the implementation of such a residual function, the filing system [Atkinson & Morrison 87] in an envisaged PPL. The PPL, X", would already provide the storage, and concurrent access to arbitrary data structures, and persistent name management. The implementation of a typical, UNIX™ -like filing system then requires very little code. PPLs also require incremental binding mechanisms which meet requirements F5 [Atkinson *et al.* 88a].

The early implementations of PPLs did not attempt to cover the fundamental functions of an operating system. The implementations, e.g. versions of PS-algol, Amber, Poly, Galileo, etc. fall short of this approach. In each case they are implemented **on top of** an operating system (invariably UNIX™), and lose much in potential efficiency by duplication and interface traversal costs. They also fall short in another respect. These languages have depended on the surrounding operating system for many functions, e.g. process creation, and have not provided such functionality themselves. Therefore, they are incomplete and, even if implemented properly, would not provide sufficient primitive functions from which to provide or implement the operating system functions F2 to F6. It may also be argued, that the example languages cited have not been targeted at the system writing process, and hence have too many high level constructs, whereas a balance towards efficiency oriented constructs [Wirth 88] would be more appropriate. This is a moot point, but is separate from the discussion here, since we consider the persistent programming language to be based on any appropriate language design with orthogonal persistence. The inclusion of polymorphic processes within the value space of Napier [Morrison *et al.* 89a, b, c] is a significant step towards providing sufficient primitive functions.

Another group of experiments concern the design of appropriate computer architectures to support persistent systems [Cockshott 88a, 89, Rosenberg *et al.* 89, Pose 89]. In principle, these experiments, which explore new architectures, would build their own operating system and persistent languages. Consequently, they might explore the structure outlined above. However, so far they have not been able to do this, as building the experimental hardware, and getting a minimal system operational has consumed the available effort. Perhaps the project most advanced (in this sense) is the implementation of Leibnitz on the Monads machine [Keedy & Rosenberg 89]. Many of the experiments also seek to build hardware which will support the targets of modern operating systems better. For example, to have much larger address spaces, to deal with distribution, to allow smaller units of protection and store to be economically supported, etc. Such goals will benefit systems where persistence is the lowest complete system supported, as the goals and loads that motivate those changes in operating systems also

apply to persistent languages, as explained above.

The arguments given above (sections 7 and 8) suggest that the functions and load on a persistent system and an operating system are sufficiently similar, that we should experiment with new relationships between them.

An operating system and any PPL will only have the same functionality if they have the same target. Each operating system designer has in mind a particular universe of applications {R} (see section 5). Similarly each language designer has a target universe {R}. Only when these two are the same will the two systems require similar functionality. For both languages and operating systems the initial universe of application is usually simple and well defined, though the definition may not be made explicit. Subsequently, partly via the misunderstandings of users, and partly via inconsistent enhancement, the definition often becomes blurred.

The operating system will not then totally disappear. Some of its kernel's implementation technology will be relocated, after a check that it is essential and appropriate, or reimplemented, in the run time system of the PPL. These primitives will be presented within the PPL, so they may be used by all programmers. The rest of the operating system contains either useful or essential superstructure. That superstructure would then be implemented in the PPL using those primitives. Experience with operating system design would then be utilised, in the specification, design and implementation of modules and procedures providing that superstructure from persistent libraries.

This new architecture has some similarities to research into lightweight operating systems where the rest of an operating system has then been implemented on top of the lightweight system.

10 Persistent languages supporting database systems

It was argued [Hepp 83] that the persistent language should provide all the central functions (concurrency, transactions, stable store, recovery) of any DBMS for any data model. Subsequently, it has been shown that it is relatively straightforward to implement various data models using a persistent language which has the following features:

i) delayed incremental binding;
ii) an extensible type system; and
iii) a callable compiler.

Examples of such demonstrations are given in Cooper *et al.* 87, Cooper & Atkinson 87, Cooper & Qin 89, Kulkarni & Atkinson 86, Cooper 89b, Abderrahmane 89. Experiments demonstrate the ease with which a persistent language may be used to implement an object oriented DBMS [Cooper 89a, Philbrow *et al.* 89].

When considered alone, the method of implementing DBMS via persistent programming languages is justified because:

i) it amortises the cost over many DBMS of providing the data model independent (perhaps 90%) of operations (e.g. recovery, locking and concurrency etc.), by these operations being implemented in the language, which is then used to implement many DBMS;

ii) it has been reported that operational prototypes of a new data model with reasonable interfaces have been implemented in as little as one month [Cooper & Atkinson 87];

iii) the DBMS is then portable, since the implementor of the persistent programming language provides the same abstract machine via each implementation; and

iv) quite high level data structures, naming systems, transaction mechanisms and concurrency arrangements will become common to more than one DBMS, thus facilitating interworking and comparison.

If the superstructure of the operating system were also built using a PPL (section 9), then it will have common underlying behaviours and structures and hence more compatible semantics with the persistent programming language, and with the DBMSs implemented in it. Therefore, we may expect interworking between the DBMSs and operating system to be made easier.

If persistent programming languages perform as expected, new application systems would be implemented in them. The various DBMS and operating systems would still be useful for two reasons:

i) to accommodate invested effort (e.g. implemented systems, existing skills); and

ii) to behave as libraries of related functions of established utility (abstract data types) for use in the construction of new applications.

The hypothesis that efficient DBMS can be built using a good quality PPL as foundation should be tested for production work. It is based on the argument that suitable optimisations may be written within the language, for example, repeated binding is avoided by calling the compiler, and optimisations, such as transformation of expressions may be performed when preparing the parameters for such a call of the compiler. Data structures, such as indexes can be built using the PPL, or built into the PPL.

In the interim we see two advantages to using PPLs to implement DBMSs based on various data models:

i) it is sufficiently easy to produce a reasonable prototype, that data models can be readily evaluated to verify their claims to aid in system design and construction; and

ii) the lower cost of implementing the data model, and the improved ease of interworking, may modify system design and construction behaviour, as people may now prefer to build one application using two data models, each suited to different parts of the application.

This whole area is ripe for further research and experimentation, as better quality PPLs become available.

11 The consequences of a persistent approach

The persistent programming paradigm provides a new way of looking at applications programming and computational problems. To illustrate this, we look at object oriented database (OODB) research and present the persistent programming approach to their situation. OODBs are chosen as our example for several reasons:

i) they have avowed goals in common with PPLs of making the relationship between program and 'database' simpler, and of improving applications programmer productivity;

ii) we have examined several OODBs recently;

iii) OODBs are a particularly active area of research at present; and

iv) some of the OODBs we examined still present mismatch problems.

A particular application area, or some other system area could equally well have been used for illustration.

11.1 The properties of OODBs

To allow our discussion to proceed, we present a description of OODBs based on papers by Bancilhon [Bancilhon 88, Atkinson *et al.* 89b]. The list of requirements for OODB is then:

i) *Encapsulation* - an object has an interface specifying a set of operations and a hidden state which may be manipulated and accessed only by that restricted set of operations;

ii) *Common storage* arrangements for data and program - these are stored using the same mechanisms and the bindings between them are also stored;

iii) *Identity* - leading to object sharing - an object may be a value in arbitrarily many other objects which store references to it, and updates are visible to the object via all references to it;

iv) *Types* - objects with the same characteristics are said to have the same type - this usually means the same signature where signature is the set of name:type pairs of the operations in the interface;

v) *Classes* - these are the set of currently extant instances of a particular object type;

vi) *Generators* - these are the operations that generate new objects - often named in terms of the type or class, e.g. **new** <class name>;

vii) *Inheritance* - this allows type specialisation;

viii) *Delayed binding* - names of operations maybe reused (overloading), the object on which an operation is being applied is identified by interpreting the name in the context of that object's type (or super-types);

ix) *Bulk operations* - usually these operations are over sets;

x) *Longevity* - any data in an OODB (including program) may be required for an arbitrarily long time;

xi) *Reliability* - data should be protected against failures by redundancy and recovery mechanisms;

xii) *Protection systems* - data needs protecting against various forms of misuse;

xiii) *Sharing and concurrency* - different requirements will be met from the same data body (requiring the equivalent of views) and uses will occur concurrently;

xiv) *Large data volumes* - the stored volume of data may range from moderate to very large scales;

xv) *Distribution* - geographically dispersed implementations of an OODB are required:

xvi) *Ad hoc use* - query languages have proved advantageous with relational systems, their role must be met for OODB; and

xvii) *Tool sets* - large collections of tools are needed to a)examine and maintain metadata, b) to examine, instrument and maintain data and program, and c) to accelerate applications system construction.

These requirements on an OODB correspond either to modelling requirements, or to database requirements. As we have stated, the modelling issue is orthogonal to persistence, and all the database requirements are requirements that could be placed on a persistent system. We first discuss a general approach to OODBs and then briefly consider the specific approach taken by the persistent paradigm to each OODB requirement.

11.2 Persistent object oriented languages

One way of achieving an OODB is to construct a persistent OOPL. For example, to make C++, objective -C, Modula-3 or Oberon persistent.

This approach has the advantage that it eliminates a reappearance of the mapping problems which occur otherwise. For example, in many of the current OODBMS, the database (i.e. classes) are defined using an OO language specific to this DBMS, but the operations (methods) are defined using some generally available programming language (e.g. C). In consequence within the method there is a mapping between the C data structures and types and those of the OODB, reintroducing the old map programming and maintenance problems.

11.3 Persistent paradigm substitutes

For each of the requirements for OODB given above we now consider the persistent programmers' options, assuming the persistent language is not object oriented. In fact, these options depend on the particular persistent language in some cases, and the language Napier88 [Morrison *et al.* 89] meets most object oriented programming requirements directly, and the process modelling available with Napier probably supersedes object oriented modelling.

i)* *Encapsulation* - most PPLs provide a modern type system (e.g. in Napier: first class procedures, ADTs and processes) which provide several appropriate mechanisms for encapsulation.

ii)* *Common storage* for program and data - most PPLs (e.g. PS-algol, Napier88) have ways to make procedures persist as properly formed closures. The constructs of Napier listed above all provide this.

iii)* *Identity* - programming languages support this as references and recursive data types. Persistent programming languages concentrate much of their implementation effort on supporting it persistently.

iv)* *Types* - these are fundamental to all existing and likely PPLs.

v) *Classes* - these are not normally maintained automatically. If a persistent programmer requires such constructs for a particular type, then an ADT is constructed which maintains the extent and provides the iterator. The bulk types of languages such as Leibnitz and Machiavelli facilitate such coding. Polymorphic ADTs, as in Napier88, may allow its generic provision. Insertion into the extent is arranged by including this in the operation to create an instance, exported from the ADT. A removal operation, exported from the ADT, would remove from the bulk type. Retention based on reachability is normally the only straightforward semantics, consequently removal may not result in the object ceasing to exist and simultaneity of the two events cannot be achieved.

vi)* *Generators* - these are automatically provided as a consequence of defining a type in most PPLs, and are usually identified by the type name.

vii) *Inheritance* - this is not generally provided in PPLs. In Napier the equivalent effects are provided by various polymorphic constructs (e.g. **env**). We discuss this design issue below.

viii)† *Delayed binding* - the naming of operations via an object and its supertype chain is not used in Napier. Equivalent effects can be achieved via environments [Dearle 89], union types, or bounded parametric polymorphism [Cardelli & Wegner 85].

ix) *Bulk operations* - the best way of providing these is not yet determined; they are present in some PPLs and have to be coded by a programmer and provided in a library in others.

x)† *Longevity* - the requirements (and achievements to date) are identical.

xi)† *Reliability* - the requirements are identical, and the support system is expected to meet them.

xii)† *Protection* - various capability structures in the support system, combined with compiler enforced encapsulation and type checking, provide security.

xiii)* *Sharing and concurrency* - Leibnitz provides views of modules. Napier, Leibnitz and χ all provide processes, and interprocess communication.

xiv)† *Large data volumes* - the requirements are similar.

xv) *Distribution* - in both contexts this is only a topic for research and is not yet available in a 'product'.

xvi) *Ad hoc use* - browsers meet one aspect of this [Dearle *et al.* 89], and immediate evaluation of expressions may meet other requirements [Atkinson & Buneman 87]. Other user interfaces to higher level views of the data would be constructed as part of the application.

xvii) *Tool sets* - the requirements are similar in both systems.

Those items marked with * are well developed concepts in programming languages, those marked † are well developed in persistence research or databases.

11.4 Inheritance and classes reconsidered

The process model in Napier provides the modelling power of some object systems with inheritance [Morrison *et al.* 89b]. The various uses of inheritance that have been proposed are clearly incompatible, for example, the use of it as a modelling construct, and as a type checking construct conflicts [Wegner & Zdonic 88]. We therefore propose that using separate constructs, as in the present persistent languages, is likely to be more efficacious, and better understood by programmers. We see the uses of inheritance as:

i) as a shorthand notation and aid to program correctness;
ii) as a modelling tool;
iii) as a means of organising nested extents;
iv) to promote code re-use;
v) as a structure for large systems; and
vi) as an abstraction over a class of data structures.

Each of these will be considered in turn. Some were discussed in earlier papers [Buneman & Atkinson 86, Atkinson & Buneman 87].

11.4.1 Shorthand notations

When a type, e.g. *person* is declared, it may be quite complex. Later, if an *employee* type is declared, the definition for person may be reused, e.g. **type** *employee* **is** *person* **and** . . . Such an abbreviation has several advantages:

i) it saves rewriting the common definition;
ii) if the *person* definition has already been demonstrated correct and useful, this is carried forward; and,
iii) if the *person* definition is revised and this source code is reprocessed, then the *employee* type may be similarly revised.

Note that none of these require the generated types to have a *recognised* relationship within the system. These effects can all be produced by suitable program development tools, such as browsers which would allow the *person* type to be found, and then 'cut' into the new code and textually extended. Such operations are supported by data dictionaries, which may also keep track of the derivation sequence and hence dependencies.

11.4.2 Modelling and inheritance

Another possible implication of a statement like **type** *employee* **is** person . . . is that the behaviour of instances of *employee* will be related to the behaviour of instances of *person*, e.g. that every operation allowed on an instance of *person* is also allowed on an instance of *employee.* A stronger interpretation is that every instance of *employee* is also an instance of *person* (so with this interpretation a reference requiring a *person* referend could have an *employee* referend). Alternatively, for every *employee* instance, there could be a corresponding *person* instance, so that a reference could discriminate between them.

Most useful models on a computer reflect dynamic behaviour. In this case we would expect to model a *person* becoming an *employee,* and also ceasing to be an *employee.* Further, we would expect movement into and out of many categories, e.g. *person* → *pupil* → *student* → *applicant* → *undergraduate* → *postgraduate* → *tutor* → etc. Such movement does not form a sequence, many developments occur independently, e.g. the above person may also perform *child* → *orphan* → *spouse* → *parent* and *person* → *golfer* → *diver* → *hillwalker.* Note also that previous states are not necessarily abandoned when a new one is entered, e.g. not all *divers* give up *golf.*

The variety of mechanisms for describing these models is further complicated by the fact that categories themselves are created (and perhaps cease) dynamically. For example, dynamically we may introduce the new specialisation of *person, speleologist*, and then derive *cave-diver* from both *speleologist* and *diver.*

If then the modelling view is taken there are many variations in the form of model programmers wish to employ. Possibly different forms for each application. It is preferable, therefore, to provide simpler primitives in the support system, out of which appropriate 'inheritance' models can be built. There are several experiments using PS-algol which show this is possible [Philbrow *et al.* 89, Cooper *et al.* 87, Kulkarni & Atkinson 86, Cooper & Atkinson 87]. It has several advantages:

i) a variety of models can coexist in the system;
ii) supporting tools or methods can facilitate the approach;
iii) programmers not using the model(s) do not have to be aware of such complex semantics; and
iv) the interference between modelling and type checking is avoided.

11.4.3 Nested extents

In languages with classes, there is a corresponding extent maintained, either as a set or sequence. Operations may be provided to iterate over that extent. Without inheritance this raises some semantic difficulties, e.g. explaining to the programmer which types have extents (**for each** *r* **in** *real* **do** . . .), which instances are in the extent (when an object has ceased to be reachable via other routes does it drop out of an extent, if an object is explicitly removed from an extent but still a referend from some other object, does it pop back in, or isn't the extent all the existing instances), what order is the iteration performed, how do creations and 'deletions' during the iteration affect that iteration, etc.? Is existence defined for this site, this database, this user's view, etc.? Some of these problems can be avoided by not having implicit extents.

When inheritance is introduced the semantics of extents becomes more complex. Is the extent of employee a subset (subsequence) of the extent of person or do they have separate extents? In the latter case, is the iteration order over one of the extents the same as that over the other? Is there a performance or order difference between **for each** *e* **in** *employee* **do** . . . and **for each** *p* **in** *person* **where** *p* **is a** *employee* **do** . . .?

Further we observe that other instances of bulk types may be useful to programmers. These may be defined separately as a built-in construct, with only a few of the difficulties above, or they may be built out of primitives by the programmers, and possibly be provided in libraries.

To provide the programmer with such an independent bulk type seems to offer more power, potential flexibility, and avoids complex semantics. The persistent programming language should therefore take this approach. Where a type related extent is needed it should be achieved by encapsulating the type in a polymorphic ADT which provides for the maintenance of extents. This is also potentially more efficient, as the extent overhead is only incurred when it will be used.

11.4.4. Code re-use

Code re-use in this context depends on inclusion polymorphism [Cardelli & Wegner 85]. For example, it means that if the operation marry had been defined for person x person , then that code could be used to marry two employees.

In most inheritance implementations this requires that the programmers have anticipated this need and defined an appropriate node in the type lattice. For example, if the database held information about dogs, horses and people, we may later find that there is a need for code to record procreation, e.g. **proc** *give-birth* [**type** *T*] (*mother: T* $\rightarrow$ *T*). Unless the system designer has a type node *animal* above *dog, horse* and *person* and below *object,* this is not definable by inclusion polymorphism based on an explicit type lattice. It is not usually possible to retrofit *animal* into the type lattice. Once inserted, it must then have the necessary attributes, e.g. *date of birth* & *gender.* Many systems then given a mother, e.g. of type *person,* produce a less precisely typed result, e.g. *animal.* Bounded universal polymorphism, or extension polymorphism, as in Machiavelli, avoids these problems, but has similar implementation costs.

11.4.5 Large system structure

Objects with inheritance claim to provide encapsulating and naming regimes to impose structure on large systems and to allow incremental systems construction through dynamic binding.

To fit all system structure within the object oriented model can sometimes be difficult. For example, there is no natural home for dyadic operations between two types, or transfer functions between types. Transfer functions present further problems as to make it possible to write them, too much of the internal structure of at least one type must be exposed.

Alternative structures combined with objects overcome these problems. First class procedures can be identified and stored to provide dyadic operations. Modules can implement families of types, exporting the types and the transfer operations without exposing internal detail.

Evolution is only partially accommodated by the signature matching of bounded universal polymorphism, as it allows code written to operate on new types which have the appropriate properties. We utilise extensible universal unions to introduce such new types and to limit checking (**env** and **any** in Napier). The **env** type also provides management of names and dynamic structures. This approach means that the programmer can choose where to pay the price of dynamic and incremental change, so such overheads are not needlessly invoked for stable parts of the system.

11.4.6 Data structure encapsulation

Whatever model is chosen to implement the functions carried by inheritance in object oriented systems, quite complex data structures result. Unaided, a programmer would find these confusing and make mistakes. Inheritance avoids that at the expense of imposing a model *a priori*.

Systematic construction methods supported by tools can overcome this as have been illustrated using PS-algol [Cooper 89a, Philbrow *et al.* 89].

11.5 Object oriented and persistent paradigms

The conclusion of the above discussion is that OO research is developing many models useful to application programmers, Zdonic's suggestions on how we may support collaboration are a good example [Fernandez & Zdonic 89].

The persistent programming research on the other hand sets out to provide a suitable set of primitives out of which to construct such models more economically. Application system building may then be a parameterisation, refinement and combination of such models, extended where necessary by new models and application specific code in the persistent language.

Development may now proceed in three ways:

i) collaboratively;
ii) with the persistent paradigm superseding the object oriented paradigm; or
iii) competitively.

If the collaborative route is taken the persistent programming paradigm provides an abstraction over the methods of providing stable stores. The object oriented community then builds libraries of abstract data types on this foundation layer, possibly exploring and developing models more rapidly, and certainly avoiding confusion between modelling and implementation technology.

If the persistent paradigm subsumes the role of object oriented systems it will do so through having high enough level constructs in its languages to satisfy application builders. In achieving this it may sacrifice some of its suitability for implementing system software.

If the two groups proceed competitively this avoids risks associated with persistent programming failing to live up to expectations. It probably delays both from achieving their expectations.

However, if both then succeed, the problems of interworking will be exacerbated, and implementation effort will be duplicated.

12 Conclusions

The persistent programming paradigm has been presented as a strong contender for greater exploration, development and use. The arguments for its use in application programming have been presented before, but they are brought together in this paper and made explicit.

The growth in persistent language research and its implementation is shown to be significant over the past ten years. In particular, there are now at least 9 persistent programming languages in use or under development, as shown in the following table:

Persistent Programming Languages

Language	Status
PS-algol	several implementations in use.
Napier88	first implementation complete.
Leibnitz	first implementation nearly complete.
χ	being implemented.
E	being implemented.
Galileo	implemented and in use.
Poly	implemented.
Amber	implemented.
Persistent Prolog	being implemented [Colomb 89].

Exploration of the addition of orthogonal persistence to a variety of other languages is recommended. To aid those who might wish to do this, we discuss the systems which support this. In most cases they now present, at some level, an abstract stable store which could be used to support a variety of languages. There are several attempts to build hardware well adapted to this purpose.

The relationship between existing persistent systems and operating systems is shown to be unsatisfactory. In particular, they represent duplicated implementation effort, and often conflict in their use of machine resource. More seriously, the combination presents an unnecessarily complex environment to applications programmers.

The overall impact of these persistent languages is illustrated by considering object oriented DBMS. One strategy would be to finesse the task of implementing object oriented DBMS by making a good object oriented language persistent. The facilities of persistent languages are compared with the provisions of OODBMS and we conclude that the existing PPLs can potentially meet all the targets of an OODBMS, and may well avoid semantic complexity inherent in the multiple uses of inheritance, classes and object based naming. We identify three strategies by which the persistent and object oriented paradigms may coexist. The most profitable is the complementary approach where persistence provides the implementation technology for object oriented systems and the object oriented paradigm provides modelling strategies for applications programming as libraries in the persistent space. We would argue that this is typical of the relationship that will develop between persistence and other database research. In particular, there will be a large repertoire of models and algorithms, including technologies now thought of as optimisation techniques, implemented within a persistent language, making available a repertoire of data handling techniques for applications implementers, as extensive as those currently prevailing for numerical work and graphics.

We identify the potential of persistent programming to be considerable. The preliminary research has been done, and the time is now ripe for major research and development projects which will bring it to much wider production use in various ways. The maximum gain will only be realised if we are prepared to totally restructure our computing systems.

Acknowledgements

This work was supported by grants from Alvey Initiative and the British Science and Engineering Research Council and by collaboration with STC Technology Ltd., at both the Universities of St. Andrews and Glasgow. GIP Altaïr have also contributed to the work at Glasgow. A grant from the British Royal Society towards travel to Australia, helped to provide the time to look at persistence more generally, and the opportunity to discuss it with enthusiasts in the field.

Bibliography

[Abderrahmane 89] **Abderrahmane, D.** Thesis in preparation, University of Glasgow, Department of Computing Science, 1989.

[Albano *et al.* 83] **Albano, A., Cardelli, L. & Orsini, R.** "Galileo: A strongly type interactive conceptual language", Tech. Rep. Internal Technical Document Services, A.T. & T. Bell Laboratories, Murray Hill, New Jersey, U.S.A., 1983.

[Albano *et al.* 85] **Albano, A., Cardelli, L. & Orsini, R.** "Galileo: A strongly typed, interactive conceptual language", *ACM Trans. Database Syst.*, 10, 2 (June 1985), 230-260.

[Amble *et al.* 79] **Amble, T., Bratbergsengen, K. & Risnes, O.** "Astral: A structured and unified approach to database design and manipulation", in *Proceedings of the Database Architecture Conference,* Venice, Italy, (June 1979).

[Argo *et al.* 87] **Argo, G., Fairbairn, J., Hughes, R.J.M., Launchbury E.J. & Trinder, P.W.** "Implementing Functional Databases", in *Proceedings of the International Workshop on Database Programming Languages,* Roscoff, France, Sept.1987.

[Atkinson 74a] **Atkinson, M.P.** "PIXIN: a network modelling language", IFIP Congress 1974, North Holland, 1974, 296-300.

[Atkinson 74b] **Atkinson, M.P.** "A survey of current research topics in data-structures for CAD", in *Programming Techniques for CAD,* ed. M.P. Sabin, NCC publications 1974, 203-218 and 297-316.

[Atkinson 75] **Atkinson, M.P.** "Network Modelling", Ph.D. Thesis, Cambridge University (1975).

[Atkinson 76] **Atkinson, M.P.** "IDL: A Machine -independent Data Language", Software Practice & Experience, Vol. 7 (1977) 671-684.

[Atkinson & Wiseman 77] **Atkinson, M.P. & Wiseman, N.E.** "Data management requirements for large scale design and production", *ACM SIGDA* Vol. 7, 1, March 1977, 2-16.

[Atkinson 78] **Atkinson, M.P.** "Programming Languages and Databases", *Proceedings of the 4th International Conference on Very Large Data Bases,* Berlin, (Ed. S.B.Yao), IEEE, Sept. 78, 408-419.

[Atkinson *et al.* 81] **Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.** "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices* Vol.17, 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

[Atkinson *et al.* 82] **Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.** "Nepal - the New Edinburgh Persistent Algorithmic Language", in *Database,* Pergammon Infotech State of the Art Report, Series 9, No.8, (January 1982) 299-318.

[Atkinson *et al.* 83a] **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).

[Atkinson *et al.* 83b] **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 *et al.* 83c] **Atkinson, M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. & Morrison, R.** "An approach to persistent programming", *The Computer Journal,* Nov 1983, Vol. 26, 4, 360-365.

[Atkinson *et al.* 84] **Atkinson, M.P., Bailey, P., Cockshott, W. P., Chisholm, K.J. and Morrison, R.** "Progress with persistent programming" in *Databases - Role and Structure* (Eds. Stocker, P.M. Gray, P.M.D. and Atkinson, M.P.) Cambridge University Press, Cambridge, England 1984, 245-310.

[Atkinson & Morrison 85a] **Atkinson, M.P. & Morrison, R.** "Procedures as persistent data objects", *ACM TOPLAS* 7, 4, (Oct. 1985) 539-559.

[Atkinson & Morrison 85b] **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.

[Atkinson *et al.* 85] **Atkinson, M.P., Buneman, O.P. and Morrison, R.** (Eds.) *Proceedings of the Persistence and Data Types Workshop,* Appin (August 1985), Persistent Programming Research Report 16, Universities of St. Andrews & Glasgow, Scotland.

[Atkinson *et al.* 86] **Atkinson, M.P., 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, N. Holland Press.

[Atkinson & Morrison 87] **Atkinson, M.P. & Morrison, R.** "Polymorphic Names and Iterations", in *Proceedings of the International Workshop on Database Programming Languages, Roscoff,* France. (eds. Buneman & Bancilhon) September 1987.

[Atkinson & Buneman 87] **Atkinson, M.P. & Buneman, O.P.** "Types and persistence in database programming languages", *ACM Surveys,* Vol.19, No.2 (June 1987) 106-190.

[Atkinson *et al.* 88a] **Atkinson, M.P., Buneman, O.P. & Morrison, R.** "Binding and Type-Checking in *Database Programming Languages",* Computer Journal, Vol. 31, No. 2 (March 1988) 99-109.

[Atkinson *et al.* 88b] **Atkinson, M.P., Buneman, O.P. & Morrison, R.** *Data Types and Persistence*, Springer Verlag, Berlin, 1988.

[Atkinson 89] **Atkinson, M.P.** Architectures for Persistent Systems, In *Proceedings 3rd International Workshop on Persistent Object Systems,* Newcastle, N.S.W., Australia, Ed. J. Rosenberg (January 1989).

[Atkinson *et al.* 89a] **Atkinson, M.P., Harper, D.J., Philbrow, P. & Watt, D.A.** "Bulk data types: design issues and practice in DBPLs", in preparation, University of Glasgow.

[Atkinson *et al.* 89b] **Atkinson, M.P. Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. & Zdonik, S.** The Object Oriented Database System Manifesto, In *Deductive Object-Oriented Databases Conference*, Kyoto, Japan (September 1989).

[Bailey 89] **Bailey, P.J.** "Performance evaluation in a Persistent Object System", in *Proceedings of the Third International Workshop on Persistent Object Systems,* January 1989, 373-385.

[Bancilhon 88] **Bancilhon, F.** "Object Oriented Database Systems" In *Proceedings ACM SIGACT-SIGMOD-SIGART, Conference on Principles of Database Systems,* Austin, Texas, May 1988.

[Bancilhon *et al.* 88] **Bancilhon, F., Barbedette, G., Benzaken, V., Debbel, C., S. Gamerman, Léclude, C., Pfeffer, P., Richard, P. & Velez, F.** The design and implementation of O_2, an object-oriented database system. In *Proceedings of the OODBS II workshop,* Bad Munster, West Germany, September 1988.

[Bancilhon & Buneman 88] **Bancilhon, F. & Buneman, O.P.** (eds.) *"The Proceedings of the International Workshop on Database Programming Languages",* Roscoff, MIT Press, 1988.

[Brown 87] **Brown, A.L.** "A distributed stable store", in *Proceedings of the Second International Workshop on Persistent Object Stores,* Appin, Scotland, 1987.

[Brown 89] **Brown, A.L.** "Persistent Object Stores", Ph.D.. Thesis, University of St. Andrews, Scotland, 1989.

[Brown & Cockshott 85] **Brown, A.L. & Cockshott, W.P.** "CPOMS - A revised version of the Persistent Object Management System in C" PPRR-13-85, Universities of St. Andrews & Glasgow, Scotland, 1985.

[Buhr & Zarnke 89] **Buhr, P.A. & Zarnke, C.R.** "Addressing in a persistent environment" in *Proceedings of the Third International Workshop on Persistent Object Systems,* January 1989, 36-50.

[Buneman *et al.* 82a] **Buneman, O.P., Frankel, R.E. & Nikhil, R.** "An implementation technique for database query languages", *ACM Trans. Database Syst.* 7, 2 (June 1982), 164-186.

[Buneman *et al.* 82b] **Buneman, O.P., Hirschberg, J. & Root, D.** "A CODASYL interface for Pascal and Ada", in *Proceedings of the Second British National Conference on Databases,* Bristol, England (July 1982) British Computer Society, London.

[Buneman 85] **Buneman, O.P.** "Data types for Database Programming", in *Data Types and Persistence* (Atkinson, M.P., Buneman, O.P. & Morrison, R. eds.), Springer-Verlag, Berlin, 1988, 91-100.

[Buneman & Atkinson 86] **Buneman, O.P. & Atkinson, M.P.** "Inheritance and Persistence in Database Programming Languages"; *Proceedings ACM SIGMOD Conference 1986,* Washington, USA, May 1986.

[Buneman & Ohori 87] **Buneman, O.P. & Ohori, A.** "A domain theoretic approach to higher order relations", A domain theoretic approach to higher-order relations. *In ICDT 86: International Conference on Database Theory* (Rome), Springer-Verlag, Berlin, 1987.

[Buneman & Ohori 89] **Buneman, O.P. & Ohori, A.** "The Database Programming Language, Machiavelli", In *Proceedings ACM SIGMOD Conference,* Oregon, U.S.A (May 1989).

[Cardelli 84] **Cardelli, L.** "A semantics of multiple inheritance", in *Proceedings of the Sophia-Anapolis Workshop,* Springer-Verlag, Berlin, 1984, 51-67.

[Cardelli 85] **Cardelli, L.** Amber, Tech. Rep. A.T. 7 T. Bell Labs., Murray Hill, N.J., U.S.A., 1985.

[Cardelli & Wegner 85] **Cardelli, L. & Wegner, P.** "On understanding types, data abstraction and polymorphism", *ACM Computing Surveys,* 17, 4 (December 1985), 471-522.

[Cardelli 87] **Cardelli, L.** "Quest" *In Proceedings 1st European Conference on Extending Database Technology,* Springer Verlag, Berlin, LNCS 303 (1988).

[Cardelli 88a] **Cardelli, L.** Personal communication, 1988.

[Cardelli *et al.* 88] **Cardelli, L., Donahue, J., Jordan, M.J., Kalsow, W. & Nelson, G.** Modula-3 Report, Olivetti Research Centre, Palo Alto, Ca., U.S.A., 1988.

[Cardelli & Mitchell 88] **Cardelli, L. & Mitchell, J.** "OOPSLA '88 Tutorial: Semantic Methods for Object-Oriented Languages", September 1988.

[Carrick & Cooper 87] **Carrick R. & Cooper, R.L.** *"The Proceedings of the Second International Workshop on Persistent Object Systems",* PPRR-44-87, Universities of St. Andrews & Glasgow, Scotland, 1987.

[Cockshot 83] **Cockshot, W.P.** *Orthogonal Persistence,* Ph.D. Thesis, University of Edinburgh, 1983.

[Cockshott *et al.* 84] **Cockshott, W.P., Atkinson, M.P.,**

[Cockshott 87] **Cockshott, W.P.** "Persistent programming and secure data storage", *Information and Software Technology,* Vol. 29, June 1987, 249-256.

[Cockshott 88a] **Cockshott, W.P.** "Addressing Mechanisms and Persistent Programming" in *Data Types and Persistence* (Atkinson, M.P., Buneman, O.P. & Morrison, R. Eds.), Springer-Verlag, Berlin, 1988.

[Cockshott 89] **Cockshott, W.P.** "Design of POMP - Persistence Object Management coProcessor", in *Proceedings of the Third International Workshop on Persistent Object Systems,* January 1989, 51-64.

[Cole & Morrison 82] **Cole, A.J. & Morrison, R.** An Introduction to Programming with S-algol, Cambridge University Press, Cambridge, England, 1982.

[Colomb 89] **Colomb, R.M.** Issues in the Implementation of Persistent Prolog. *In Proceedings 3rd International Workshop on Persistent Object Systems,* Newcastle, N.S.W., Australia (Jan. 1989), 67-79.

[Connor *et al.* 89] **Connor, R., Brown, A., Carrick, R., Dearle, A. & Morrison, R.** "The Persistent Abstract Machine", in *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, N.S.W., Australia , January 1989, 80-95.

[Cooper 89a] **Cooper, R.L.** "The Implementation of an Object-Oriented Language in PS-algol", In *Proceedings of the Third International .Workshop on Persistent Object Systems,* Newcastle, N.S.W., Australia, January 1989.

[Cooper 89b] **Cooper, R.L.** Thesis in preparation, University of Glasgow, Department of Computing Science, 1989.

[Cooper & Atkinson 87] **Cooper, R. L. & Atkinson, M.P.** "Requirements Modelling in a Persistent Object Store", in *Proceedings of the 2nd International Workshop on Persistent Object Systems,* Appin, Scotland. August 1987.

[Cooper *et al.* 87] **Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.** "Constructing Database Systems in a Persistent Environment", in *Proceedings of the Thirteenth International Conference on Very Large Databases,* Brighton, September 1987, 117-125.

[Cooper & Atkinson 87] **Cooper, R.L. & Atkinson, M.P.** "A Requirements Modelling Tool Built in PS-algol", *Persistent Programming Research Report* 54, Universities of Glasgow and St. Andrews, 1987.

[Cooper & Qin 89] **Cooper, R.L. & Qin, Z.** "Implementing IFO in PS-algol", in preparation, University of Glasgow, Department of Computing Science.

[Dearle 88] **Dearle, A.** *On the construction of Persistent Programming Environments,* Ph.D. Thesis, University of St. Andrews, Scotland, 1988.

[Dearle & Brown] **Dearle, A. & Brown, A.L.** Safe Browsing in a strongly typed Persistent Environment. *Computer Journal,* Vol. 31, No. 2 (March 1988), 540-544.

[Fernandez & Zdonik 89] **Fernandez, M.F. & Zdonik, S.B.** "Transaction groups: a model for controlling cooperative transactions", in *Proceedings of the Third International Workshop on Persistent Object Systems,* Jan. 1989, 128-138.

[Hall 83] **Hall, P.A.V.** "Adding database management to Ada", *SIGPLAN Not. (ACM)* 13, 3 (April 1983), 13-17.

[Hammer & McLeod 81] **Hammer, M. & McLeod, D.** "Database description with SDM: A semantic database model", *ACM Trans. Database Systems,* 6, 3 (September 81), 351-386.

[Hepp 83] **Hepp, P.E.** *A DBS Architecture Supporting Coexisting Query Languages and Data Models,* Ph.D. Thesis, University of Edinburgh, Scotland, 1983.

[Horowitz & Kemper 83] **Horowitz, E. & Kemper, A.** AdaRel: A relational extension of Ada, Tech. Rep. TR-83-218, Department of Computing Science, University of Southern California, Los Angeles, California, U.S.A, 1983.

[Hurst & Sajeev 89] **Hurst, A.J. & Sajeev, A.S.M.** "A capability based language for persistent programming: Implementation issues", in *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, N.S.W., Australia, (ed. Rosenberg, J.), January 1989, 186-201.

[Keedy & Rosenberg 89] **Keedy, J.L. & Rosenberg, J.** "Uniform support for collections of objects in a persistent environment", in *Proceedings of the 22nd Hawaii International Conference on System Sciences* (ed. B. D. Schriver), (Jan 1989), Vol. II, 26-35.

[Koch *et al.* 83] **Koch, J., Mall, M., Putfarken, P., Reid, M., Schmidt, J.W. & Zehnder, C.A.** Modula/R report - Lilith version, Tech. Rep. Institute für Informatik, Eidgenossische Technische Hochschule, Zurich, 1983.

[Krablin 85] **Krablin, G.L.** "Building flexible multilevel transactions in a distributed persistent environment", in *Proceedings of the Persistent and Datatypes Workshop,* Appin, Scotland, 1985, 86-117.

[Kulkarni 83] **Kulkarni, K.G.** Evaluation of Functional Data Models for Database Design and Use, Ph.D. Thesis, University of Edinburgh, Scotland, 1983.

[Kulkarni & Atkinson 84] **Kulkarni, K.G. & Atkinson, M.P.** "Experimenting with the Functional Data Model", in *Databases - Role and Structure,* Cambridge University Press, Cambridge, England, 1984.

[Kulkarni & Atkinson 86] **Kulkarni, K.G. & Atkinson, M.P.** "EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

[Lécluse *et al.* 88] **Lécluse, C., Richard, P. & Velez, F.** O_2, an Object-Oriented Data Model. In *Proceedings of the ACM-SIGMOD Conference,* Chicago, June 1988.

[Matthes & Schmidt 89] **Matthes, F. & Schmidt, J.W.** "The type system of DBPL", in *Proceedings 2nd International Workshop on Database Programming Languages,* Portland, Oregon, (June 1989).

[Matthews 85] **Matthews, D.C.J.** Poly manual, Tech. Rep. 63, Computer Laboratory, University of Cambridge, Cambridge, England, 1985.

[Merrett 77] **Merrett, T.H.** "Relations as programming language elements", *Inf. Process. Lett.,* 6, 1 (February 1977), 29-33.

[Morrison *et al.* 86a] **Morrison R., Dearle, A., Brown, A. & Atkinson M.P.** "An integrated graphics programming environment", *Computer Graphics Forum,* Vol. 5, 2, June 1986, 147-157.

[Morrison *et al.* 86b] **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,* 14, 3, 1986.

[Morrison *et al.* 89a] **Morrison, R., Brown, A., Carrick, R., Connor, R., Dearle, A., and Atkinson, M.P.,** "The type system of Napier", in *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, N.S.W., Australia, (January 1989), 253-270.

[Morrison *et al.* 89b] **Morrison, R., Brown, A.L., Carrick, R., Barter, C.J., Hurst, A.J., Connor, R., Dearle, A. & Livesey, M.J.** "Language design issues in supporting process-oriented computation in persistent environments", in *Proceedings of the 22nd Hawaii International Conference on System Sciences* (ed. B.D. Schriver), (Jan. 1989), Vol. II, 736-744.

[Morrison *et al.* 89c] **Morrison, R., Brown, A.L., Conner, R. & Dearle, A.** "*The Napier reference manual*", University of St. Andrews, Department of Computational Science, St. Andrews, Scotland, 1989.

[Mylopoulos *et al.* 80] **Mylopoulos, J., Bernstein, P.A. & Wong, H.K.T.** "A language facility for designing database intensive applications", *ACM Trans. Database Syst.,* 5, 2 (June 1980), 185-207.

[Ohori 87] **Ohori, A.** "Orderings and types in databases", in *Proceedings of the Roscoff Workshop on Database Programming Languages*, Altaïr, France, September 1987.

[Organick 72] **Organick, E.I.** The MULTICS System, MIT Press, Boston, Mass., U.S.A. 1972.

[Philbrow 88] **Philbrow, P.** "The mongoose benchmark", internal note, November 1988.

[Philbrow & Atkinson 88] **Philbrow, P. & Atkinson, M.P.** Exception Handling in a Persistent Programming Language, *Computer Journal,* to be published.

[Philbrow et al. 89] **Philbrow, P., Harper, D.J. & Atkinson, M.P.** "An object oriented programming methodology in PS-algol", in *Proceedings of 2nd Workshop on Database Programming Languages,* Portland, Oregon, U.S.A. (June 1989), 313-330.

[Pose 89] **Pose, R.D.** "Capability based, tightly coupled multiprocessor hardware to support a Persistent Global Virtual Memory", in *Proceedings of the 22nd Hawaii International Conference on System Sciences* (ed. B.D. Schriver), Vol. II, 36-45.

[Rosenberg *et al.* 89] **Rosenberg, J., Koch, D.M. & Keedy, J.L.** "A massive memory supercomputer", in *Proceedings of the 22nd Hawaii International Conference on System Sciences* (ed B.D. Schriver), (January 1989), Vol. I, 338-345.

[Rowe & Shoens 79] **Rowe, L. & Shoens, K.** "Data abstraction, views, and updates in Rigal", in *Proceedings of ACM SIGMOD International Conference on Management of Data,* Boston, Mass., U.S.A. (May 79), ACM, New York, 71-81.

[Schmidt 77] **Schmidt, J.W.** "Some high level language constructs for data of

type relation", *ACM Trans. on Database Syst.* 2, 3 (September 1977), 247-261.

[Shapiro 79] **Shapiro, J.E.** "THESEUS - A Programming Language for Relational Databases", *ACM Trans. Database Syst.,* 4, 4 (Dec. 79) 493-517.

[Shipman 81] **Shipman, D.W.** "The functional data model and the date language DAPLEX", *ACM Trans. Database Syst.,* 6, 1 (March 1981), 140-173.

[Smith *et al.* 83] **Smith, J.M., Fox, S. & Landers, T.** Adaplex: Rationale and reference manual, 2nd Computer Corporation of America, Cambridge, Mass., U.S.A., 1983.

[Stonebreaker 87] **Stonebreaker, M.J.**
in *Proceedings of the 13th International Conference on Very Large Databases,* Brighton, England, (September 1987).

[Trinder & Wadler 88] **Trinder, P.W. & Wadler, P.L.** "List comprehensions and the relational calculus", in preparation. University of Glasgow.

[Trinder 89] **Trinder, P.W.** "Optimisation of List Comprehensions", in preparation, University of Glasgow.

[Turner 87] **Turner, D.A.** Miranda System Manual, Research Software Ltd., Canterbury, England, 1987.

[Wadler 85] **Wadler, P.L.** An introduction to Orwell, Oxford University Handbook, December 1985.

[Wadler 87] **Wadler, P.L.** "List Comprehensions", Chapter 7 of The Implementation of Functional Programming Languages, Peyton-Jones, S.L., Prentice Hall, 1987.

[Wai 88] **Wai, F.** Ph.D. Thesis, University of Glasgow, Scotland, 1988.

[Wasserman *et al.* 81] **Wasserman, A.I., Shurtz, D.D., Kersten, M.L., van Reit, R.P. & van de Dippe, M.D.** "Revised report on the programming language PLAIN", *ACM SIGPLAN Not.* (1981).

[Wegner & Zdonik 88] **Wegner, P. & Zdonik, S.** Inheritance as an Incremental Modification Mechanism, or What Like is and isn't like, In *Proceedings ECOOP 88,* Lecture Notes in Computer Science 322, Springer-Verlag, Berlin.

[Wirth 88] **Wirth, N.** "The Programming Language Oberon", *Software: Practice and Experience,* 18, 7, (July 1988) 671-690.

[Zdonik 89] **Zdonik, S.B.** "Query Optimisation in Object-Oriented Databases", in *Proceedings of the 22nd Hawaii International Conference on System Sciences* (ed. Schriver, B.), January 1989, Vol.

DISCUSSION

During the lecture Professor Randell noted that the history of persistent programming languages seemed to have been dominated by Europeans and he wondered if this was so. In reply, Professor Atkinson disagreed and referred to work done at DEC SRC by Cardelli, together with some work that had been carried out at Waterloo.

Later Professor Nygaard questioned the assertion, made on one of Professor Atkinson's slides, that programs were equal to data. Professor Atkinson answered this by saying that he considered programs to be made up from code which was encapsulated in the form of procedures. Since procedures were first-class objects and could be assigned to variables then they were indeed simply data.

During Professor Atkinsons description of binding, Professor Randell commented that "programming was the art of premature binding". Professor Atkinson answered this by saying that tools were required to enable a programmer to decide what should be statically bound, and what should be bound dynamically.