This paper should be referenced as:

Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Munro, D. “Delivering the Benefits of Persistence to System Construction and Execution”. In Proc. 17th Australasian Computer Science Conference, Christchurch, New Zealand (1994) pp 711-719.

Delivering the Benefits of Persistence to System Construction and Execution

R. Morrison, C. Baker, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby and D.S. Munro

Department of Mathematical and Computational Sciences, University of St Andrews,
North Haugh, St Andrews, Fife KY16 9SS, Scotland.

Phone: +44 334 63254
Internet: {ron, craigb, richard, quintin, graham, dave}@dcs.st-andrews.ac.uk

Abstract

Persistent programming systems are generally recognised as the appropriate technology for the construction and maintenance of large, long-lived application systems. Many successful prototypes have been constructed, and a large body of application building experience is emerging. However, all persistent systems to date have been provided within an operating system environment, and the services provided by the operating system have been relied upon to give the necessary support for program tasks such as editing and linking. Only the execution of persistent programs has occurred within the persistent environment.

Here we examine some of the consequences of providing all the support required for program construction within the same environment as the persistent data. We show how having the construction, compilation, and execution processes all operating within a single environment leads to some powerful new techniques. This new power is achieved by the sharing of persistent data across the hitherto enclosed boundaries of these processes.

1 Persistent Systems

In recent years considerable research has been devoted to the investigation of the concept of persistence and its application in the integration of database systems and programming languages [Atk78, ABC+83]. As a result a number of persistent systems have been developed including PS-algol [PS88], Napier88 [MBC+89], Flex [Cur85, Sta86], Ten15 [CF86], Galileo [ACO85], TI Persistent Memory System [Tha86], Amber [Car85] and Trellis/Owl [SCW85]. In each of these systems persistence has been used to abstract over the physical properties of data such as where it is kept, how long it is kept and in what form it is kept, thereby simplifying the task of programming. The benefits of orthogonal persistence have been described extensively in the literature [ACC82, ABC+84, AM85, AMP86, AB87, Dea87, MBC+87, Wai87, AM88, Dea88, Bro89, MBC+89, Con90, MBC+90]. These can be summarised as

- improving programming productivity from simpler semantics;
- removing ad hoc arrangements for data translation and long term data storage; and
- providing protection mechanisms over the whole environment.

The persistence abstraction has long been recognised as the appropriate underlying technology for long lived, concurrently accessed and potentially large bodies of data and programs. Typical examples of such systems are CAD/CAM systems, office automation and CASE tools. Object-Oriented Database Systems such as GemStone [BOP+89] and O_2 [BBB+88] have at their core a persistent object store and are particular examples of persistent systems within a particular paradigm. The goal of persistence research is to allow these socially and economically important persistent application systems to become more sophisticated and more economically viable.

2 Moving into the Persistent Environment

All persistent systems developed to date are supported within standard operating systems. Programs which operate over persistent data are written using operating system facilities, such as text editors, linkers and compilers, and are then executed against the persistent environment. This is in conflict with the concept of persistence, as that part of an application which consists of source code, version information, etc. is held in a different data store with different characteristics from the rest of the data, which in many cases includes the executable code.

Although there are clear pragmatic reasons for this approach the existence of the two separate worlds has been observed to add complexity to application systems. Even in persistent systems which support first class code in the form of procedure values, complex conventions are required to map between

the source of this code in the operating system environment and the executable version in the persistent environment.

Since the initial design and implementation of the Napier88 language environment, however, much support has subsequently been developed in the form of application building tools normally provided at the operating system level. This has made possible the development of applications within the persistent environment itself. This support consists of the following tools, all of which have been developed in Napier88 itself and then provided within the Napier88 persistent store environment:

- a window manager
- a text editor, written to work within the window manager
- a persistent store browser
- a Napier88 compiler, which compiles Napier88 programs into a form suitable for execution within the persistent environment

Although the building of such support is clearly a significant task, once it has been established it is possible to build persistent application systems in a single environment. This avoids the unnecessary complexity caused by the artificial interaction between the operating system and the persistent environment. However, although achieving this continuity was the initial motivation of building these tools, the provision of such support turns out to lead to further new advantages. In this paper we concentrate on two of these:

- the provision of new paradigms for system construction, and
- new possibilities to perform optimisation of persistent code

The ability to provide the new techniques is a consequence of all phases of the software development process being integrated into the same environment.

The presence of persistent data in the software construction environment allows the introduction of new binding paradigms, in particular the ability to bind persistent values directly into both source and executable code. This allows programs that contain links to persistent values within their source code, instead of textual denotations of these links which must be evaluated and bound to the program code during or immediately before its execution. This structured form of a program bears a similar relationship to purely textual programs as hyper-text does to ordinary text, and so the new style of program is known as a hyper-program [KCC+92]. Hyper-programming is not possible unless the source and executable forms of programs reside in the same persistent space as the rest of the application data.

The optimisation architecture described relies upon source and executable forms of programs being resident in the execution environment along with the compiler. In many situations more efficient code may be generated by the compiler where some knowledge of the way in which code is used dynamically is available. In the integrated environment, dynamic information gathering may be tied to optimisation processes by which the source code is recompiled automatically and incrementally according to the use of its executable equivalent. Thus the initial static trade-offs made by the compiler on the execution profile of the program may be dynamically re-evaluated in the light of execution experience. Again, this style of optimisation is not possible where the compilation and execution environment are divorced.

3 New Paradigms for System Construction

3.1 Hyper-programming

The traditional representation of a program as a linear sequence of text forces a particular style of program construction to ensure good programming practice. Persistent systems have the ability to allow the persistent environment to participate in the program construction process. This raises the possibility of allowing the representations of source programs to include direct links to values that already exist in the environment, giving hyper-programs.

Figure 1 shows an example representation of a hyper-program. The hyper-program contains both text and links which denote data items in the persistent store. The first link is to a procedure to write out a string; this is called to write a prompt to the user. The program then calls another procedure to read in a name, and then finds an address corresponding to the name. This is done by calling a lookup procedure which is one of the components of a table package linked into the hyper-program. The address is then written out. Note that code objects (*readString* and *writeString*) are treated in exactly the same way as data objects (the table).

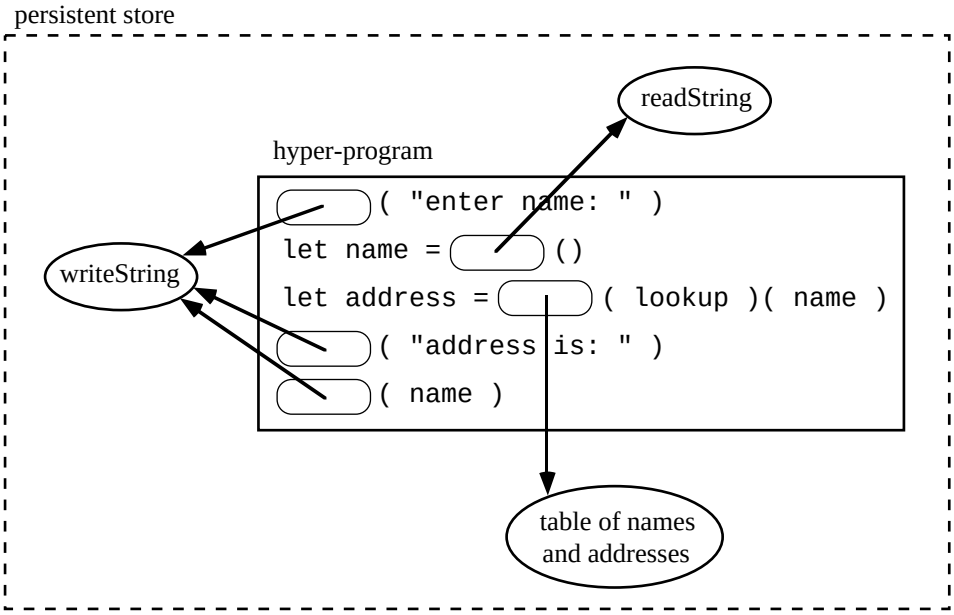

Figure 1: A hyper-program

Many programs may share links and the graph of program components can become highly interconnected. Other benefits of hyper-programming are discussed in [Kir92]. They include:

- being able to perform program checking early;
- support for source representations of all procedure closures;
- being able to enforce associations from executable programs to source programs;
- increased program succinctness; and
- increased ease of program composition.

While hyper-programming provides composition time binding, it does not disallow other binding times. Early static binding trades flexibility for safety whereas dynamic binding does the opposite. Hyper-programming systems can provide both within the same framework by allowing hyper links to both values and locations (R-values and L-values). Thus although the locations are strongly typed they may change their values providing the flexibility of dynamic binding.
The principal requirement for supporting a hyper-programming system is a persistent store to contain the program representations and the data items denoted by the links in the programs. The persistent store is stable and supports referential integrity. Hence when a reference to a data item in the store has been established, the data item will remain accessible for as long as the reference exists.

Secondly, all hyper-program representations, both source and executable, must consist of denotable values within the persistent programming language environment. Thus the compiler can arrange for compiled programs to contain links to their source representations which are themselves values in the persistent store. A further consequence is that the compilation process itself must also be supported within this same environment. One mechanism particularly well suited to realise this is known as type-safe linguistic reflection, as described in [SSS+92].

A third requirement is for tools which provide the programmer with a graphical representation of the persistent store. The representation shows the values, locations and types in the persistent store and the links between them. The programmer can point to the representations of specific data items and obtain tokens for them to be incorporated into hyper-programs.

A hyper-programming system also has to support additional facilities for ‘programming in the large’, that is, building large applications from smaller components. These include facilities for controlling the sharing of components between applications, for limiting the visibility of some components for protection reasons, and for imposing a degree of partitioning on the persistent store to aid intellectual manageability and execution efficiency. A model to support these facilities is the *hyper-world*.

3.1.1 Hyper-worlds

There are a number of facilities that a persistent programming environment should support if it is to provide for the software engineering process as a whole. These include:

- program composition, compilation and execution;

- storing of source and compiled versions of programs;
- debugging;
- documentation;
- decomposition of large application programs into components, and organisation of those components;
- navigating the persistent store to locate programs and other data with given attributes;
- querying of the types of programs and data in the persistent store;
- facilities for exporting completed application systems once development has been completed; these should allow implementation details to be hidden, and allow control over whether the identity of components linked to by the application should be preserved.

The model of hyper-programming allows source programs to contain links to any other data in the persistent store. In large scale systems this generality may lead to several problems. Firstly, the store may become intellectually unmanageable as the number of links increases. Secondly, evolution of application programs by substituting new versions of their components becomes difficult to manage if unrestricted linking to the components is permitted—it may be necessary to locate each data item linked to the component being substituted and determine whether a new version of the data item is required in turn. In addition the model must provide a uniform framework for storing meta-data about application components.

A research topic is the provision of additional structure over a basic hyper-programming system to address these needs. The *hyper-world* model offers the programmer a loose coupling mechanism to offset the disadvantages of the tight coupling made possible by hyper-programming. In this model, based in part on that described in [WA86], the persistent store is partitioned into a number of application spaces or hyper-worlds. Each hyper-world contains the program components and data used by an application, and a schema that describes their relationships. Each hyper-world has a single visible component which may be linked to from outside the hyper-world.

The schema includes documentation information, a type description and hyper-program source for each component. It also includes a representation of the component linking topology, and a list of type definitions local to the hyper-world. This allows the programmer to perform various queries over the components and to determine the implications of replacing a component with a changed version.

The partitioning supported by hyper-worlds will reduce to a manageable scale the complexity of problems such as keeping track of inter-component links. This may be done by restricting the region of interest from the entire persistent store to the hyper-world. The partitioning may also allow type-checking to be performed more efficiently and act as the unit of optimisation for store operations such as garbage collection. Figure 2 shows a representation of a persistent store containing nested hyper-worlds and linked components.

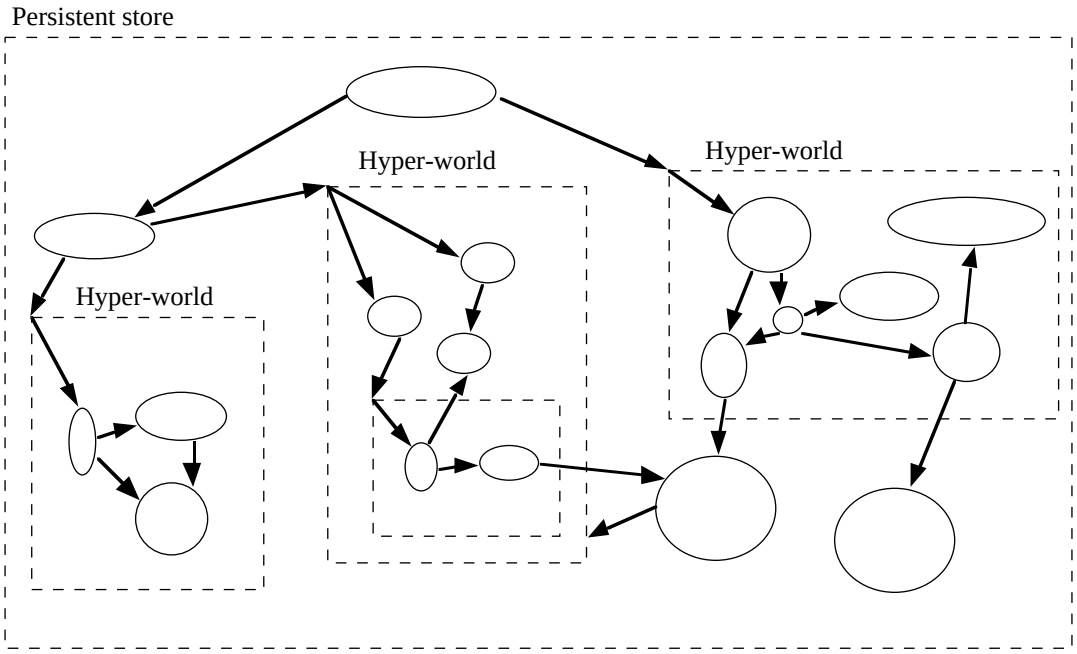

Figure 2: A persistent store with hyper-worlds

In the hyper-world, the hyper-programming model can be re-used to yield new facilities. For example, in traditional version control and configuration management systems, a naming convention is used to identify the system components. In the hyper-world the components can be identified by direct link, either R-value or L-value, and the need for a naming convention is eliminated and along with it the source of many errors.

The open questions for the construction and utilisation of a hyper-world are:

- What are the required set of tools to support design, construction, maintenance and operation?
- How can incremental change be accommodated?
- How can hyper-worlds be exported and imported to other persistent environments?
- What requirement does a hyper-world place on the supporting languages and stores?

3.2 Optimisation of Persistent Systems

The second aspect to supporting the entire software life cycle within the persistent environment is to improve system performance by utilising the ubiquitous availability of the persistent store. This yields a unique opportunity for optimisation by using the persistent store as a cache for information that can be shared by all system components. Of particular interest is the symbiotic relationship of the compiler, run-time system and object store in a persistent environment.

Compilers use static analysis to generate code which is intended to be, by some measure, efficient. Compilers can also pass on hints to the run-time system and the object store as to the probable dynamic use of the system. For example, the compiler may detect that certain groups of objects are always required together. This information can be used by the object store to cluster objects.

In traditional systems, the compilation system is forced to make efficiency decisions under *a priori* rules. There is usually no mechanism for dynamic information to be re-introduced into the compilation system. In persistent systems, however, the run-time system can record dynamic profile information in the persistent store. The compiler can then be called to recompile programs using this information where it is advantageous to do so. For example, code generation usually entails a trade off between the execution speed of the generated code and its size. Where this statically calculated trade-off is shown to be disadvantageous to the overall system performance, the run-time system can invoke the compiler to produce more efficient code according to the dynamic profile.

Here we outline a general system architecture for optimisation within persistent systems and concentrate on two examples of the technique. The first is that of discovering locality in persistent systems and the second is generating efficient implementations of polymorphic code.

3.2.1 A System Architecture for Optimisation

The general optimisation architecture enhances the operation of some aspect of system behaviour by using the persistent environment as a cache to record the dynamic behaviour of the system associated with that aspect. The information is then analysed by enhancement programs and used as the basis for possible optimisations. Recording, analysis and optimisation take place in four stages:

- The executing program records information about its dynamic execution in the persistent store.
- Using the cached information the enhancer program determines whether a potential optimisation exists. Optimisation involves altering trade-offs associated with the implementation.
- If an optimisation exists then the enhancer determines whether the optimisation should be performed. The decision is made using a cost function that may operate over both static and dynamic information and determines whether the benefits of the optimisation outweigh the cost of making it. The enhancer may invoke the compilation system.
- The optimisation is applied to the persistent environment if the cost function returns a positive result.

Subsequent executions will benefit from any optimisation that is made.

It should be pointed out that this general technique has to be applied with some care. The cost of recording dynamic profile information and invoking the enhancer may outweigh the benefits of the optimisation. There are, however, many fairly obvious places that the technique will work, and indeed where it will not, and working experience is required to discover other areas of applicability.

One dimension of the architecture is when the optimisation is performed. Dynamic optimisation in database systems is not usual and the conventional wisdom is to recompile/optimise when some exceptional event occurs, such as dropping an index. Such circumstances are accommodated within this framework but it is also interesting to note that a spectrum of possible optimisation times exists thereby tempting experimentation.

Optimisations may be made to both code and data within the persistent store. Below two examples will be used to illustrate the architecture. The two examples are discovering locality, and therefore optimising data utilisation, and the code generation of polymorphic functions as an example of code optimisation. Such optimisations can also be applied by this technique to any data or meta-data within the persistent store.

3.2.2 Locality

In common with all Virtual Memory systems and Database Management systems it has been observed that increasing the locality of persistent data can enhance the overall system performance. Locality is important for clustering and garbage collection as well as the exploitation of distribution.

Traditionally the discovery of locality information by static analysis has yielded only very limited gains. In long running systems with shared data, the locality of data changes with time and usage. Many programs that share data often have conflicting locality profiles. This has led to research into such areas as dynamic clustering techniques but also with limited success.

The optimisation architecture may be used to discover the locality of data and to distributed the data in the following way. Information on the locality of data and the pattern of use of data is recorded in the persistent store and shared among system components. Thus a garbage collector may use the locality information when compacting a store. A dynamic clustering mechanism may use the information in a similar manner and a scheduler may use it to distribute programs in both space and time. Garbage collectors and clustering mechanisms are examples of enhancement programs.

The above technique for dynamic clustering is already used in some database systems. Persistent systems with their close relationship of the compiler, run-time system and object store can go further.

One approach to locality is to encourage the user to organise the data in a manner that allows the system to discover locality. It is here that the use of hyper-worlds is important. Where a program is constructed as a hyper-program the compiler can discover which hyper-links are used together. This information is passed to the run-time system and the object store. Hyper-programs also encourage the user to organise programs in localities, the hyper-worlds. Information on the use of the data in a hyper-world is recorded in the hyper-world and again used by the compiler, run-time system and object store for optimisation.

Information is also recorded on the use of meta-data. For example, a particular editing session may always be followed by a compilation or a particular data set may always be used in combination with a version control system. Some of this information is user specified and some discovered dynamically. As far as we know no previous attempt has been made to capture such information and use it to enhance system performance.

3.2.3 Implementing Polymorphism

Polymorphism in a programming language provides the ability to write programs that are independent of the form of the data objects that they manipulate. Thus it provides an abstraction over the form of the data which is often categorised by type.

Polymorphism may be expressed at many levels of abstraction depending on the style of implementation. The following three categories represent extremes in the possible range of implementation techniques [MDC+91]:

- Textual polymorphism. In this category polymorphism is only expressed at the source code level. Different executable code may be produced for each different specialising type. The execution efficiency of the equivalent monomorphic procedure will be achieved using this technique. This is the optimum speed efficiency. However textual polymorphism may be expensive in terms of the storage space required for the specialised forms. An example of this kind of polymorphism is found in the generics of Ada [DOD83].
- Uniform polymorphism. Both the source code and the executable code are independent of particular specialising types in this category and so only a single executable version of a polymorphic expression is required. This is achieved using a single representation for all data. The use of space for polymorphic code forms is optimal. However, the uniform data format has efficiency implications for non-polymorphic data values in terms of both space and time. This kind of polymorphism is found in ML [MTH89].
- Tagged polymorphism. In this category uniformity is expressed both at the source and executable levels but non-uniform data formats are used. Thus the executable code for a polymorphic expression is parameterised in some way by type information describing the representation of data values being manipulated. Effectively every data item is tagged with its type. Code space is again optimised but all values have to pay the price of the tagging which is expensive if performed in software and is generally not available in hardware. Tagged

polymorphism is used in the implementation of some object-oriented languages [GR83].

These implementation techniques represent different trade-offs between run-time efficiency and the space required for polymorphic code forms. A particular trade-off is traditionally determined statically during the code generation phase of compilation and is therefore fixed for the lifetime of the procedure. In persistent systems it is possible to mix the implementation strategies for polymorphism. The run-time system can record the manner in which the polymorphic code is used and invoke the compiler to generate a more efficient form. Thus, for example, the uniform implementation may be replaced by a textual implementation where appropriate. This, of course, is not always possible but the case analysis is a known result.

It is also possible for mixed representations of polymorphic code to co-exist. Indeed the first attempt at using the code may discover that only the source is available in the store as the compiler decided not to translate it. The compiler could then generate a more efficient form of the polymorphic code for initial use. On discovering the pattern of use of the code even more efficient forms can be generated [Cut92]. For example, it may be discovered that 90% of the calls of a polymorphic procedure specialise it to a particular type. The concrete form for that type could be generated and used to form the 90% of the calls in conjunction with the polymorphic form that is used for the 10% of the calls.

The improvement of polymorphic code is an example of the enhancement of code generation.

4 Current Status

A prototype version of the hyper-programming system exists, supporting the following:

- an integrated compilation, execution and browsing system;
- hyper-program bindings to values, locations and types in the persistent store;
- automatic retention of procedure source code as hyper-programs in the persistent store; and
- the ability to browse procedure closures and to bind values in them into hyper-programs.

The hyper-world facilities, under construction, will provide a means of structuring applications constructed in the integrated hyper-programming environment. They will give flexible control over the visibility of data, which in turn will support the protection mechanisms necessary to regulate access to procedure closures and other data.

The general optimisation architecture has been designed and implemented. The trade-offs, cost functions and optimisation techniques of the architecture are dependent on the particular optimisation to be carried out. The architecture has been tested with the two examples:

- the discovery of locality, and
- the implementation of polymorphism.

The modular structure of the layered Napier88 persistent object store and the compilation system have been found to minimise the programming effort required.

5 Conclusions

The consequences of providing support for program construction within the same environment as the persistent data have been found greater than initially expected. Having the construction, compilation, and execution processes all operating within a single environment leads to some powerful new techniques which may be used to facilitate many aspects of the software development life cycle. This new power is achieved by the sharing of persistent data across the hitherto enclosed boundaries of these processes. Two particular example of this have been shown in some detail:

- the presence of persistent data within the program construction and compilation environment has been used to achieve new program construction paradigms that are not otherwise possible; and
- the presence of the execution and compilation systems in the same environment as the source and executable programs has been used to achieve a new and powerful style of optimisation architecture.

Both of these new developments are believed highly significant in themselves. Perhaps even more interesting is the fact that they have both been developed in a very short time from the establishment of the completely integrated persistent system, which suggests that many other possibilities for the improvement of software engineering environments may result from this integration.

6 Acknowledgements

This work was supported by ESPRIT III Basic Research Action 6309 – FIDE$_2$. The original hyper-programming research was carried out in conjunction with Alan Dearle and Alex Farkas of the University of Adelaide.

Since the development of hyper-programming in the Napier88 environment, we have discovered that

the concept, if not the name, was present in the Flex system developed at RSRE (DRA) on the ICL Perq [Sta86]. In subsequent discussions with Ian Currie, the main designer, it is clear that the same motivations and outlook were present in his system which was Algol 68 based.

7 References

[AB87] Atkinson, M.P. & Buneman, O.P. "Types and Persistence in Database Programming Languages". ACM Computing Surveys 19, 2 (1987) pp 105-190.

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. "An Approach to Persistent Programming". Computer Journal 26, 4 (1983) pp 360-365.

[ABC+84] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. "Progress with Persistent Programming". Universities of Glasgow and St Andrews Technical Report PPRR-8-84 (1984).

[ACC82] Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. "PS-algol: An Algol with a Persistent Heap". ACM SIGPLAN Notices 17, 7 (1982) pp 24-31.

[ACO85] Albano, A., Cardelli, L. & Orsini, R. "Galileo: a Strongly Typed, Interactive Conceptual Language". ACM Transactions on Database Systems 10, 2 (1985) pp 230-260.

[AM85] Atkinson, M.P. & Morrison, R. "Procedures as Persistent Data Objects". ACM Transactions on Programming Languages and Systems 7, 4 (1985) pp 539-559.

[AM88] Atkinson, M.P. & Morrison, R. "Types, Bindings and Parameters in a Persistent Environment". In **Data Types and Persistence**, Atkinson, M.P., Buneman, O.P. & Morrison, R. (ed), Springer-Verlag (1988) pp 3-20.

[AMP86] Atkinson, M.P., Morrison, R. & Pratten, G.D. "A Persistent Information Space Architecture". In Proc. 9th Australian Computing Science Conference, Australia (1986).

[Atk78] Atkinson, M.P. "Programming Languages and Databases". In Proc. 4th IEEE International Conference on Very Large Databases (1978) pp 408-419.

[BBB+88] Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lécluse, C., Pfeffer, P., Richard, P. & Valez, F. "The Design and Implementation of O_2, an Object-Oriented Database System". In **Lecture Notes in Computer Science 334**, Dittrich, K.R. (ed), Springer-Verlag (1988) pp 1-22.

[BOP+89] Bretl, B., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., Williams, M. & Maier, D. "The GemStone Data Management System". In **Object-Oriented Concepts, Applications, and Databases**, Kim, W. & Lochovsky, F. (ed), Morgan-Kaufman (1989).

[Bro89] Brown, A.L. "Persistent Object Stores". Ph.D. Thesis, University of St Andrews (1989).

[Car85] Cardelli, L. "Amber". AT&T Bell Labs, Murray Hill Technical Report AT7T (1985).

[CF86] Core, P.W. & Foster, J.M. "Ten15: An Overview". RSRE Malvern Technical Report 3977 (1986).

[Con90] Connor, R.C.H. "Types and Polymorphism in Persistent Programming Systems". Ph.D. Thesis, University of St Andrews (1990).

[Cur85] Currie, I.F. "Filestore and Modes in Flex". In Proc. 1st International Workshop on Persistent Object Systems, Appin, Scotland (1985) pp 325-334.

[Cut92] Cutts, Q.I. "Delivering the Benefits of Persistence to System Construction and Execution". Ph.D. Thesis, University of St Andrews (1992).

[Dea87] Dearle, A. "Constructing Compilers in a Persistent Environment". In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987).

[Dea88] Dearle, A. “On the Construction of Persistent Programming Environments”. Ph.D. Thesis, University of St Andrews (1988).

[DOD83] “Reference Manual for the Ada Programming Language”. U.S. Department of Defense Technical Report ANSI/MIL-STD-1815A (1983).

[GR83] Goldberg, A. & Robson, D. **Smalltalk-80: The Language and its Implementation**. Addison Wesley, Reading, Massachusetts (1983).

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag (1992) pp 86-106.

[Kir92] Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1992).

[MBC+87] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment”. Software Engineering Journal, December (1987) pp 199-204.

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. University of St Andrews Technical Report PPRR-77-89 (1989).

[MBC+90] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Dearle, A., Rosenberg, J. & Stemple, D. “Protection in Persistent Object Systems”. In **Security and Persistence**, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 48-66.

[MDC+91] Morrison, R., Dearle, A., Connor, R.C.H. & Brown, A.L. “An Ad-Hoc Approach to the Implementation of Polymorphism”. ACM Transactions on Programming Languages and Systems 13, 3 (1991) pp 342-371.

[MTH89] Milner, R., Tofte, M. & Harper, R. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts (1989).

[PS88] “PS-algol Reference Manual, 4th edition”. Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[SCW85] Schaffert, C., Cooper, T. & Wilpot, C. “Trellis Object-Based Environment Language Reference Manual”. DEC Systems Research Center Technical Report 372 (1985).

[SSS+92] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. & Alagic, S. “Type-Safe Linguistic Reflection: A Generator Technology”. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/49 (1992).

[Sta86] Stanley, M. “An Evaluation of the Flex PSE”. RSRE Malvern Technical Report 86003 (1986).

[Tha86] Thatte, S.M. “Persistent Memory: A Storage Architecture for Object Oriented Database Systems”. In Proc. ACM/IEEE International Workshop on Object-Oriented Database Systems, Pacific Grove, California (1986) pp 148-159.

[WA86] Wile, D.S. & Allard, D.G. “Worlds: An Organizing Structure for Object-Bases”. In Proc. 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Palo Alto, California (1986).

[Wai87] Wai, F. “Distribution and Persistence”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987) pp 207-225.