This paper should be referenced as:

Connor, R.C.H., Morrison, R., Atkinson, M.P., Matthes, F. & Schmidt, J. “Programming in Persistent Higher-Order Languages”. In **Euro-ARCH'93**, Spies, P.P. (ed), Springer-Verlag (1993) pp 288-300.

Programming in Persistent Higher-Order Languages

Connor R.C.H., Morrison R., Atkinson M.P.†,
Matthes F.‡ and Schmidt J.W.‡

Department of Mathematical and Computational Science,
University of St Andrews, St Andrews, Fife, Scotland.

†Department of Computer Science, University of Glasgow,
Glasgow G12 8QQ, Scotland.

‡Fachbereich Informatik, Universität Hamburg, Vogt-Kölln Straße 30,
D-2000 Hamburg 54, Germany.

{richard, ron}@dcs.st-and.ac.uk
mpa@dcs.glasgow.ac.uk
{matthes, schmidt}@dbis1.informatik.uni-hamburg.de

Abstract

Persistent programming languages differ from traditional languages in that data of arbitrary lifetimes is fully governed by the type system. Such languages lead to radically different programming paradigms and methodologies for two important reasons:

- the high-level type system may be relied upon as a protection mechanism even for long-term data
- data of any type, including first-class procedures and abstract data types, may be kept for arbitrary lifetimes

The combination of these means that the kind of sophisticated typing commonly used in programs which operate over short-term data may be extended to all data manipulated by a long-lived system. This paper exposes some of the ways in which well-known type system features may be powerfully used in contexts normally associated with operating systems and database management systems.

1. Introduction

Information systems frequently have to deal with longevity and scale in the data that they support. Data is required with lifetimes which match the real-world processes which they represent, which can range from microseconds to years. Data is also commonly required to support the work of large organisations, with time scales of up to hundreds of years.

In an orthogonally persistent programming system, the manner in which data is manipulated is independent of its persistence. The same mechanisms operate on both short-term and long-term data, avoiding the traditional need for separate systems to control access to data of different degrees of longevity. Thus data may remain under the control of a single persistent programming system for its entire lifetime. The benefits of orthogonal persistence have been described extensively in the literature and 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.

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 Pascal/R [Sch77] PS-algol [PS88], Napier88 [MBC+89], DBPL [MS89], Galileo [ACO85], TI Persistent Memory System [Tha86], Amber [Car85] , Trellis/Owl [SCW85] and Tycoon [MS92]. The persistence abstraction is widely 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, CASE tools and software engineering environments. Object-Oriented Database Systems such as GemStone [BOP+89] and O_2 [BBB+88] have at their core a persistent object store. Process modelling systems use a persistent base to preserve their modelling activities over execution sessions [BPR91]. The goal of persistence research is to allow these socially and economically important persistent application systems to be more sophisticated and more economically viable.

This paper surveys some programming methodologies and styles that have been developed after the extensive use of persistent languages in the construction of medium scale software systems. Section 2 introduces the main point of underlying technology which makes the techniques possible: that of an enforced, persistent type system. Section 3 shows how the sophisticated protection offered by many modern type systems can be used to great effect when extended to cover long-term data. Section 4 surveys hyper-programming, a new style of programming in which the program source itself contains bindings to typed values in the persistent object store. The advantages of hyper-programming are not limited to persistent systems; however the persistent technology is a requirement for the safe implementation of a hyper-programming system.

2. Type systems and persistence

Type systems are historically viewed as mechanisms which impose static safety constraints upon a program. Within a persistent environment, however, the type system takes on a wider role.

Data manipulated by a programming language is governed by that language's type system. In non-persistent languages, however, data which persists for longer than the invocation of a program may only be achieved by the use of an operating system interface which is shared by all applications. As a consequence of this, such data passes beyond the jurisdiction of the type system of any one language.

Mechanisms which govern long term data, such as protection and module binding, must be dealt with at the level of this interface. Historically this has the consequence that the type system may not be enforced, and knowledge of the typed structure of data may not be taken advantage of. This is shown diagramatically in Figure 2.1.

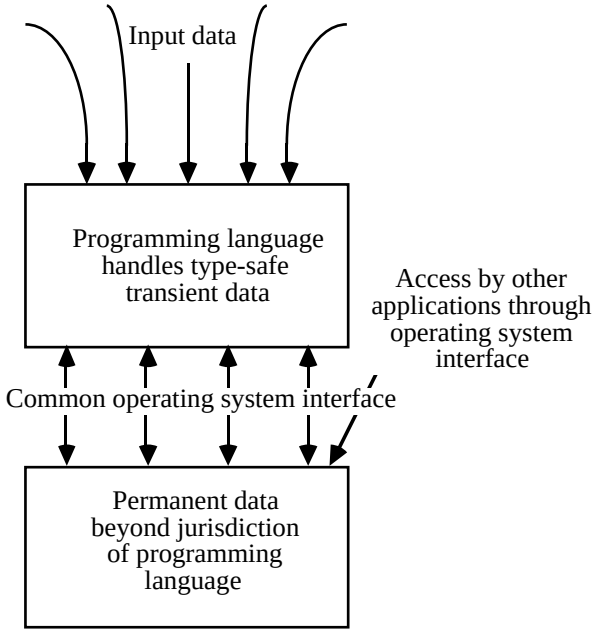

Figure 2.1 Traditional strategy for permanent and shared data

In a persistent system, the storage of data beyond a single program invocation is handled by programming language mechanisms, and no common operating system interface is necessary. The only route by which data may be accessed is through the programming language, and so the type system of a single language may be used to enforce protection upon both transient and permanent data. High-level modelling may be relied upon for the entire lifetime of the data, as it never passes outside the language system. Figure 2.2 gives a diagrammatic view of such a system.

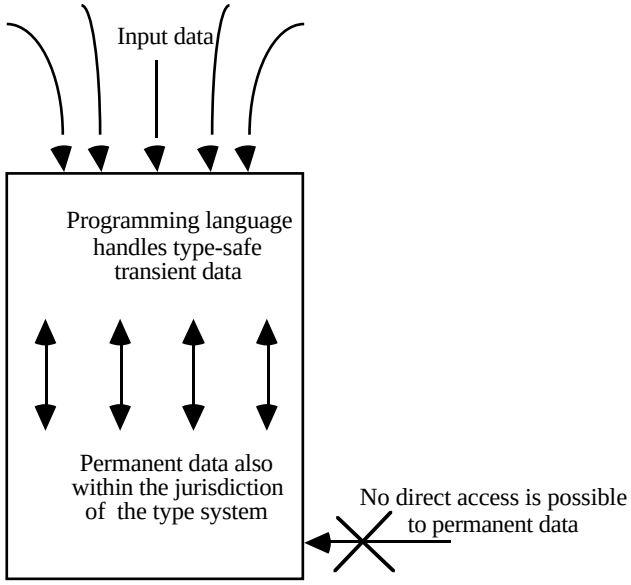

Figure 2.2 The persistent strategy

The universality of the persistent type system has consequences in terms of both the modelling and protection provided by the type system. With respect to modelling, the persistent type system must be sufficiently flexible to allow the modelling of activities normally provided by untyped support systems. Such activities include, for example, the linking of separately prepared program units, and file system access protection. With respect to protection, the increased role of a type system means that any protection mechanisms

programmed at a high level may be fully relied upon to protect the data for its lifetime, as access from outside the constraints of the type system is not possible. In particular, various high level information hiding techniques may be used to restrict data access, instead of relying upon the normally coarse grain and typeless control provided by outside technologies.

3. Programming with persistent type systems

Traditionally, computer systems rely upon the type systems of individual programming languages as the only protection mechanism for transient data, and provide a number of different mechanisms, such as file system protection or database constraints, for permanent data. This is because transient and persistent data exist in two separate universes, with different operations available upon them. In persistent programming systems, however, there is no clear difference between transient and permanent data, and this two-level protection scheme is not an appropriate model.

The provision of type-safe permanent data provides one kind of control that is not available in a non-persistent model. Type systems by themselves, however, lack two important categories of integrity protection. These are access protection, to prevent data from being accessed by programs which do not require it, and integrity constraints, which are normally finer-grained that those which may be modelled by type systems.

Both of these categories of protection may be explicitly programmed in many programming languages. However, in a non-persistent system, such programmed protection may not be extended to data exported from a program. In a persistent system, however, data storage is performed entirely within a programming language, and there is no problem with the storage and sharing of any kind of data. Access protection and integrity constraints may therefore be flexibly programmed according to the needs of the individual data.

Access protection may be programmed in a persistent system by the use of constructs which allow information hiding. These include inclusion polymorphism, abstract data types, and higher-order procedures. What these have in common is that they may hide part of the state of a program by introducing state which is not directly denotable. This makes it impossible to specify another program which will access the hidden state.

There are three well-known mechanisms which allow the programming of information hiding within a strong type system. These are subtyping, procedural encapsulation (1st-order information hiding) and existential data types (2nd-order information hiding). Subtyping achieves protection by removing type information, causing the static failure of programs which may try to perform undesirable accesses. 1st-order information hiding prevents the protected data from being named by an untrusted program, allowing access only through a procedural interface. 2nd-order hiding is somewhere between these two, allowing access mainly through procedures, but also allowing the protected data to be named. This data is, however, viewed through a mechanism which causes type information loss, and therefore allows only a limited set of operations to be performed on it.

3.1. Subtyping

In general, systems which allow subtyping allow any data value to be used in place of one typed with less functionality. One type is a subtype of another if all operations allowed on the

second type are also allowed on the first. In the most general form of subtyping, often referred to as inclusion polymorphism, it is type correct for the use of any value to be replaced by the use of any of its subtypes.

A number of different semantics are possible for the definition of a subtype rule. Here the semantics of Cardelli [Car84] are adopted, using structural type equivalence and an implicit subtyping rule.

Subtype inheritance is usually regarded as a general modelling technique. In particular it allows the declaration of procedures which operate over any type with at least a set of required properties. However, using an object as one of its supertypes is also equivalent to hiding some of the functionality which the object possesses. For example, the following introduces the names *employee* and *person* as record types:

```
type employee is record( name , address : string ; salary : int )
type person is record( name , address : string )
```

Type *employee* is a subtype of type *person*, and an object of type *employee* may be substituted in any context where an object of type *person* is expected. This would have the effect of hiding the salary field by the loss of type information. If another user is only to be allowed this restricted access to employee objects, this view of the object may be exported, for example by use of an explicit type coercion:

```
let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
let exportJoe : person = joe
```

In this *joe* is declared to be an object of type *employee* with the given field values. *exportJoe* is of type *person*, denoted by the type after the ":" symbol, but has the value *joe*. This means that a user of the value *exportJoe* will now have the value of the original record. However, it is not possible to express an operation to access the salary field of this value due to the restrictions of the static type system. That is, the *salary* field cannot be used with the object *exportJoe* since such a program would fail during static analysis.

In a non-persistent system such protection appears to be of little use. The point of this mechanism applied to persistence, however, is that arbitrary typed values may be made to persist, and the type system strictly enforces any future access to them by other programs. In the above program segment the identifiers *joe* and *exportJoe* denote the same value, but are of different types. It is then possible to set up two different contexts in which these differently typed denotations of the same value are made persistent. Programs which subsequently bind to the *exportJoe* context will be able to use the value only as a type *person*. Subtying of persistent data therefore provides the essence of a database viewing mechanism when applied to persistent data.

3.2. 1st-order information hiding

Access to data can also be restricted by only allowing access to procedures which are defined over the data, and not allowing the data itself to be visible. This is a common model for abstract data types, and is known as 1st-order information hiding [CW85]. It may be achieved in a number of ways but it will be described here in terms of a language which has first-class procedure values and block-style scoping. Access to the original data may then be

removed simply by its identifier becoming unavailable. For example, the following type defines a *Person* as a record containing procedures which define three operations:

```
type Person is record
(
        getName,
        getAddress      : proc( → string ) ;
        putAddress      : proc( string )
)
```

This allows a finer grain of restriction than that achieved by subtype inheritance, in that the name and address may be read, but only the address may be changed. Access to the data may be removed by placing its declaration in a block so that its identifier is lost from scope after the *Person* object has been constructed. This is shown in Figure 3.1. The exported procedures, which have the data encapsulated within their closures, are then the only way in which the original value may be accessed. Again, this relies upon the static properties of the system to prevent the access since a program which attempts direct access to *joe* will fail statically by the scoping rules of the language.

```
let exportJoe =
begin
        let joe = employee( "Joe Doe", "1 Assignment Boulevard", 100000 )
        Person(     proc( → string ) ; joe.name ,
                    proc( → string ) ; joe.address ,
                    proc( new : string ) ; joe.address := new       )
end
```

Figure 3.1 Hiding the Data Representation

In Figure 3.1, *exportJoe* is declared to have the value obtained by executing the block. This is a structure of type *Person* with three procedure fields. Each procedure uses the object *joe* which is inaccessible by any other means after exit from the block.

Further flexibility is possible using encapsulation in that dynamic properties may be specified, and access may be denied dynamically if required. For example, perhaps there exists an integrity constraint that an address may not be more than 100 characters long. This can be programmed in the procedural encapsulation, as shown in Figure 3.2. The only difference here is that the *putAddress* procedure checks the dynamic constraint, and raises an exception if it is not met.

```
let exportJoe =
begin
        let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
        Person(     proc( → string) ; joe.name,
                    proc( → string) ; joe.address,
                    proc( new : string ) ;
                        if length( new ) ≤ 100
                            then joe.address := new
                            else raise longAddress( new )      )
end
```

Figure 3.2 Refining the Interface

A particular example of a dynamic constraint allows access to the original data to be protected by password. A procedure can be provided in the interface which will return direct access to a user with sufficient privilege. Figure 3.3 shows the extended definition required,

with an extra procedure in type *extraPerson* which returns the representation of the data only if it is supplied with a string equal to the password used to create it. In this situation, the programmer responsible for constructing the view of the data will have enough information to extract the representation. Alternatively, it would be possible to arrange system-wide passwords which would decide whether access is allowed or not.

```
type extraPerson is record
(
        getEmployee     : proc( string → employee ) ;
        getName,
        getAddress      : proc( → string ) ;
        putAddress      : proc( string )
)

let exportJoe = proc( password : string → extraPerson )
begin
        let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
        extraPerson(    proc( attempt : string → employee )
                            if attempt = password
                                then joe
                                else failValue,
                        proc( → string ) ; joe.name,
                        proc( → string ) ; joe.address,
                        proc( new : string ) ; joe.address := new)
end
```

Figure 3.3 Protection by Password

This technique gives a result not dissimilar from the kind of module provided by Pebble [BL84] and a high level language analogy of capabilities [DvH66].

Instead of a string password, an unforgeable “software capability” may be used. It is a consequence of orthogonal persistence that, for any type over which identity is defined, the identity of any value is unique for the lifetime of the system. Therefore the identity of such a value is unforgeable. If such a value is bound to the closure of a procedure as its password, then to use the procedure a programmer must somehow have access to the same value.

3.3. 2nd-order information hiding

2nd-order information hiding does not restrict access to the protected values, but instead abstracts over the type of the protected value to restrict operations allowed on it. Thus the protected values may be manipulated for some basic operations, such as assignment and perhaps equality, but their normal operations are not allowed due to the type view. This allows the representation objects themselves to be safely placed in the interface along with the procedures which manipulate them.

One mechanism which allows 2nd-order information hiding is the existential data type as described in [MP88]. This allows the definition of interface types which are abstracted over. As names for these types are declared before the existential type definition, different parts of the definition may be bound to the same type. As before, only the basic operations defined on all types are allowed over the abstracted types, but values which are abstracted by the same name are statically known to be compatible. *Person* as above may be redefined as:

```
type Person is absType[ absPersonType ]
(
        absPerson,mum,dad,favourite     : absPersonType ;
        getName,getAddress              : proc( → string ) ;
        putAddress                      : proc( string )
)
```

The identifier in square brackets before the body of the type declaration declares a name for a type which is abstracted over. This allows a tighter definition of such types, as it can now be seen where the same type appears in the interface. Components of the same instance of an interface may be bound together by a **use** clause. For example,

```
use exportJoe as joe in
        joe.favourite := joe.mum
```

may be statically determined to be type correct, as the *favourite* and *mum* fields must be type compatible to allow the object to be created.

This static binding of equivalent types may also be used to allow the interface procedures to be defined over the type of the hidden representation. A more flexible definition which allows the name and address operations to be performed on any of the people in the interface would be:

```
type Person is abstype[ absPersonType ]
(
        absPerson,mum,
        dad,favourite           : absPersonType ;
        getName,getAddress      : proc( absPersonType → string ) ;
        putAddress              : proc( absPersonType,string )
)
```

This allows the definition of n-ary operations over the hidden representation type. For example, a procedure may be placed in the interface which tests if two people have the same address:

```
type Person is absType[ absPersonType ]
(
        absPerson,mum,
        dad,favourite   : absPersonType ;
        getName,
        getAddress      : proc( absPersonType → string ) ;
        putAddress      : proc( absPersonType,string ) ;
        sameAddress     : proc( absPersonType,absPersonType → bool )
)
```

This example illustrates a major difference in power between 1st-order and 2nd-order information hiding. With 2nd-order, a type is abstracted over, and procedures may be defined over this type. With 1st-order hiding, it is the object itself which is hidden within its procedural interface. Procedures which operate over more than one such object may not be defined sensibly within this interface. Therefore any operations defined over two instances must be written at a higher level, using the interface. At best this creates syntactic noise and is inefficient at execution time. It also means that such operations are defined in the module which uses the abstract objects, rather than the module which creates them. Some examples are not possible to write without changing the original interface.

Once again it should be stressed that the type constructs introduced here are well understood and found in many programming languages. The use of these mechanisms in persistent languages relies upon the ability of values of these types to be kept for arbitrary lifetimes, and the binding of new programs to them to be strongly typed. Therefore the protection provided by the mechanism can be used in a much wider context.

4. Hyper-programming

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 which 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, Kir92, Cut92].

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.

The primary motivation for providing a hyper-programming system is to allow the programmer to compose programs interactively, navigating the persistent store and selecting data items to be incorporated into the programs. The programmer has the option of linking existing data items into a program by pointing to graphical representations. This removes the need to write access specifications for persistent data items that are accessed by a program. The ability to link to data items at run-time is still required in the cases where data becomes available only after a program is written.

Figure 4.1 shows an example representation of a hyper-program. The hyper-program contains both text and links that denotes 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.

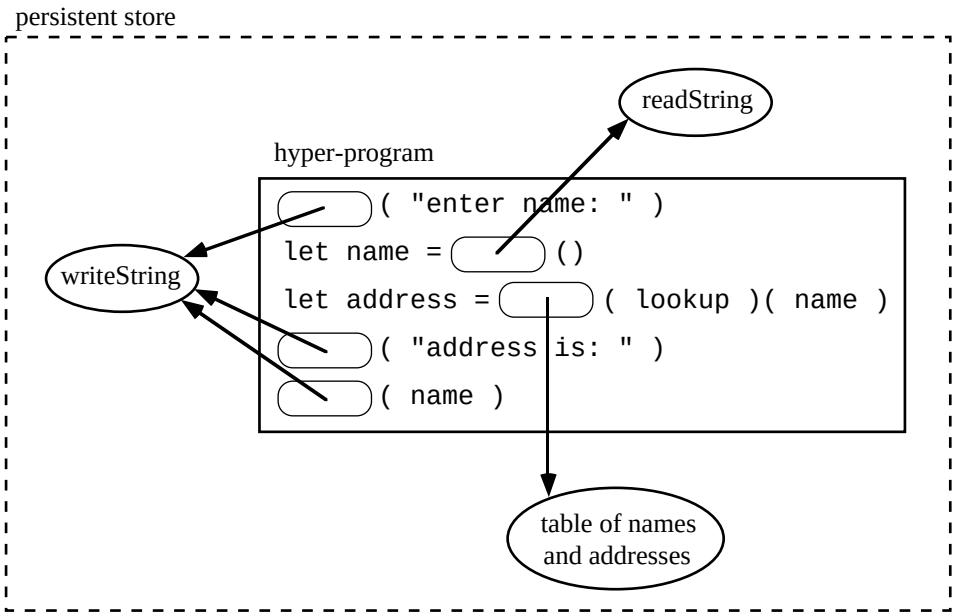

Figure 4.1 A hyper-program

Many programs may share links and the graph of program components can become highly interconnected. Other benefits of hyper-programming include:

- increased ease of program composition;
- being able to perform program checking early;
- being able to enforce associations from executable programs to source programs;
- availability of an increased range of linking times;
- reduced program verbosity; and
- support for source representations of procedure closures.

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 must be stable, and support 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. An incremental binding mechanism is also required, to allow newly constructed programs to be bound back into the system within which they were constructed.

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. Thus an implementation requires to be based not only upon a persistent programming language, but one in which full support is available for the entire software lifecycle. This includes for example a program editor, an object browser, a compiler and a window manager, all written in the persistent language itself.

A prototype hyper-programming system is available in the Napier88 release system [KCC+92]. It is important to note that hyper-programming is not an intrinsic part of the Napier88 language, and could be provided in any persistent language given the appropriate support technology.

5. Conclusions

It has been shown how the provision of persistence sheds new light upon the use of type system features common in non-persistent languages. The fact that attributes of a type may

be absolutely relied upon for the lifetime of the data it describes gives some powerful new methodologies for the programming of activities normally associated with untyped, operating system support systems.

The mechanisms of subtype inheritance, procedural encapsulation, and existential data types have been discussed with relation to the programming of protection within a persistent system. These mechanisms may be used to program protection only in a persistent system; in a non-persistent system the protected data is always susceptible to abuse by the common operating system interface through which its storage must be arranged.

Protection in database systems is normally provided by viewing mechanisms. These allow database programmers to set up different interfaces over the same data, so that users with different privileges may perform different operations. It is clear that the information hiding techniques presented above may be used to program flexible viewing mechanisms within a persistent programming system.

The last consequence of strongly typed persistent data presented is the ability to safely allow the style of programming known as hyper-programming. The concept of hyper-programming is straightforward: programs may contain direct links to values, rather than textual denotations which are bound to at or after compilation. The only way this programming style can be supported with any reasonable degree of safety is by the system being contained in a stronly-typed persistent environment, so that the types of the direct bindings can be enforced.

6. Acknowledgements

Much of this work was supported by ESPRIT II Basic Research Action 3070 – FIDE, and SERC grant GR/F 02953. Richard Connor is supported by SERC Postdoctoral Fellowship B/91/RFH/9078. The hyper-programming system was partly conceived and wholly implemented by Quintin Cutts and Graham Kirby of St Andrews University.

7. References

[ABC83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. "An Approach to Persistent Programming." The Computer Journal, 26, 4, (1983), 360-365.

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

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

[BBB+88] Bancilhon, F., Barbedette, G., Benzaken, B., Delobel, C., Gamerman, S., Lecluse, C., Pfeffer, P., Richard, P. & Valez, F. “The Design and Implementation of O2, an Object-Oriented Database System”. In **Lecture Notes in Computer Science 334**, Springer-Verlag (1988) pp 1-22.

[BL84] R. Burstall and B. Lampson "A Kernel Language for Abstract Data Types and Modules" Proc. International Symposium on the Semantics of Data Types LNCS Vol. 173, Springer - Verlag (1984)

[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).

[BPR91] Bruynooghe, R.F., Parker, J.M. & Rowles, J.S. “PSS: A System for Process Enactment”. In Proc. 1st International Conference on the Software Process: Manufacturing Complex Systems (1991).

[Car84] L. Cardelli "A Semantics of Multiple Inheritance" Proc. International Symposium on the Semantics of Data Types LNCS Vol. 173, Springer - Verlag (1984) pp 51 - 67

[Car85] Cardelli, L.
Amber. Tech. Report AT7T. Bell Labs. Murray Hill, U.S.A. (1985).

[CW85] L. Cardelli and P. Wegner "On Understanding Types, Data Abstraction and Polymorphism" ACM Computing Surveys 17, 4 (December 1985) pp. 471 - 523

[Cut92] Kirby, G.N.C “Delivering the Benefits of Persistence to System Construction and Execution” Ph.D. Thesis, University of St Andrews (1992).

[DvH66] J.B. Dennis and E.C. van Horn "Programming Semantics for Multiprogrammed Computations" CACM 9, 3 (1966) pp 143 - 145

[KCC92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. "Persistent Hyper-Programs". 5th International Conference on Persistent Object Systems, Italy (1992).

[KCC+92] Kirby, G.N.C., Cutts, Q.I., Connor, R.C.H., Dearle, A. & Morrison, R. “Programmers’ Guide to the Napier88 Standard Library, Edition 2.1”. University of St Andrews (1992).

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

[MBC89] Morrison, R., Brown, A.L., Connor, R. & Dearle, A.
"The Napier88 Reference Manual". Universities of Glasgow & St Andrews Persistent Programming Research Report 77-89 (1989)

[MP88] J.C. Mitchell and G.D. Plotkin "Abstract Types have Existential Type" ACM ToPLaS 10, 3 (July 1988) pp. 470 - 502

[MS89] F. Matthes and J.W. Schmidt "The Type System of DBPL" In R. Hull, R. Morrison and D. Stemple (editors) Proc. 2nd International Workshop on Database Programming Languages Morgan - Kaufmann (1989) pp 219 - 225

[MS92] Matthes, F. & Schmidt, J.W. “Definition of the Tycoon Language TL - A Preliminary Report”. University of Hamburg Technical Report 062-92 (1992).

[PS88] "PS-algol Reference Manual".
Universities of St Andrews and Glasgow PPRR39 (1988).

[Sch77] J.W. Schmidt "Some High-Level Language Constructs for Data of Type Relation" ACM ToDS 2, 3 (1977) pp 247 - 261

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

[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.