This paper should be referenced as:

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

Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment

R.Morrison, A.Brown, R.Carrick, R.Connor, A.Dearle & M.P.Atkinson[+]

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

[+] Department of Computing Science,
University of Glasgow,
Lilybank Gdns.,
Glasgow,
Scotland G12 8QQ.

Abstract

The major requirements of a system for software reuse are that it must provide an abstraction mechanism for adequately describing the components; a mechanism for naming and storing the components; and a mechanism for composing new objects out of existing components.

In this paper we describe a polymorphic type system that may be used to describe generic components and a persistence mechanism that may be used to name, store and compose components. By integrating the two, we obtain a strongly typed persistent environment that is an ideal base as a system for software reuse.

1 Introduction

The most cost effective way to build a system is to construct it out of existing components[5]. This has been a major motivation in our efforts to design and implement a persistent information space architecture[4] which can be viewed as an enabling technology for Integrated Project Support Environments[15]. The persistent store can be used as a place to deposit system components which can be reused later in the construction of new systems.

There are three major issues that we will focus on in this paper with regard to software reuse in a persistent environment. They are

- a powerful type system to describe all system components,
- a mechanism for controlling the naming of objects, and
- a method of binding, for composing systems out of reusable components.

A powerful type system is required so that all the data objects in the system can be represented by a type. The advantage of this is that there is one mechanism, the type checker, used to ensure the legal use of all compositions of system components. The type system must also have powerful abstraction mechanisms which allow specialisation for a particular reuse. In this way the abstraction is only written once, while specialisation may be used many times with the resultant saving in software costs.

The second facility for reusability is a mechanism for controlling the use of names in a system. Components are identified by name, and the ease of use of a system partly depends on finding the correct name. In a system with a large number of names it should be possible to interrogate the system so that the correct component can be found.

Finally, the method of composing systems out of reusable components should be simple and, if possible, one of the binding mechanisms already used in the language. This binding

mechanism is necessarily dynamic since we wish to construct systems incrementally out of components of a live system. The only major difficulty with this is in integrating it with a strong type checker.

Here we describe our work with the language Napier[2,17], which combines a powerful polymorphic type system with persistence to provide the above mechanisms for a strongly typed object oriented environment.

2. Types

The Napier type system is loosely based on one suggested by Cardelli & Wegner[7]. All data objects in the system can be described by the following rules

1. The scalar data types are integer, real, boolean, string, pixel, picture, file and null.
2. The type image is the type of an object consisting of a rectangular matrix of pixels.
3. For any data type t, *t is the type of a vector of with elements of type t.
4. For identifiers I_1,...,I_n and types t_1,...,t_n, **structure**(I_1:t_1,...,I_n:t_n) is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n.
5. For identifiers I_1,...,I_n and types t_1,...,t_n, **variant**(I_1: t_1,...,I_n: t_n) is the type of a variant with options I_i and corresponding types t_i, for i = 1..n.
6. For any data types t_1,...,t_n and t, **proc**(t_1,...,t_n -> t) is the type of a procedure with parameter types t_i, for i = 1..n and result type t. The type of a resultless procedure is **proc**(t_1,...,t_n).
7. For any procedure type, **proc**(t_1,...,t_n -> t) and type identifiers T_1,...,T_m, **proc**[T_1,...,T_m] (t_1,...,t_n -> t) is the type **proc**(t_1,...,t_n -> t) universally quantified by types T_1,...,T_m. These are polymorphic procedures.
8. For identifiers I_1,...,I_n and types t_1,...,t_n, **env** is the type of an environment with fields I_i and corresponding types t_i, for i = 1..n.
9. For any type identifiers W_1,...,W_m, identifiers I_1,...,I_n and types t_1,...,t_n, **abstype** [W_1,...,W_m] (I_1: t_1,...,I_n: t_n), is the type of an existentially quantified data type. These are abstract data types.
10. The type **any** is the infinite union of all types.
11. For any user-constructed data type t and type identifiers, T_1,...,T_n, t[T_1,...,T_n] is the type t parameterised by T_1,...,T_n.

The universe of discourse of the language is defined by the closure of rules 1 and 2 under the recursive application of rules 3 to 11.

We will describe the more important aspects of this type system by example. The essential element for software reuse is that there should be a high degree of abstraction. Thus, in the above, vectors and structures are regarded as store abstractions over all data types, procedures as abstractions over expressions and statements, abstract data types as abstractions over declarations and polymorphism as an abstraction over type. The infinite unions **env** and **any** are used to support persistence as well as being a general modeling technique. The types picture, pixel and image are used for graphics.

Allowing such abstract forms in the object space enables the programmer to store them and to specialise them on reuse. This reduces the total amount of code required in any system, since we only have to write them once and may reuse them many times. We will return to a discussion of the power of these abstractions when we have studied the other aspects of the type system.

The underlying philosophy of the Napier type system is that types are sets of values[7], and therefore type equivalence is decided by structure. We will assume that readers are familiar

with most aspects of such a type system and concentrate on the parts relevant to software reuse, starting with polymorphism[12].

2.1 Polymorphism

Polymorphism is a mechanism whereby we can abstract over type. It is perhaps best explained by an example; in Napier

let int_id = **proc** (x : int -> int) ; x

This defines a procedure called int_id that takes an integer as a parameter and returns it as the result. This is easily recognised as the identity procedure for integers. The type of the procedure is written in Napier as **proc** (int -> int).

We may also wish such an identity procedure for reals and which we could write as

let real_id = **proc** (x : real -> real) ; x

Polymorphism allows us to combine the above two procedures into one by abstracting over the type. This can be written in Napier as

let id = **proc** [t] (x : t -> t) ; x

The square brackets signify that the procedure is parameterised by a type and that once given that type, the procedure is from type t to t. To call this procedure we may write

```
id [int] (3)   which yields 3 or
id [real] (4.2)      which yields 4.2
```

or we may use the type parameter by itself

id [int] which yields the equivalent procedure to int_id above.

Thus we have written one procedure, id, which in fact is an infinite number of identity procedures, one for each type as it is parameterised. We use square brackets for type parameters to signify that types are not part of the value space of the language but are based on the philosophy that types are sets of values.

The type of id may be written as ∀t.**proc** (t -> t) since for every type t, it acts as a procedure that will take a parameter of type t and return a value of that type. The procedure is said to be universally quantified by t. ∀t.**proc** (t -> t) is written as **proc** [t] (t -> t) in Napier. Procedures of these polymorphic types are first class and may be stored, passed as parameters and returned as results. This method of polymorphism, sometimes called parametric polymorphism, first appeared in the language ML [13], and later in Russell [8] and Poly [11].

The advantage of the polymorphic abstraction should be obvious in the context of software reuse. We may, for example, write a procedure to sort a vector of integers and another procedure to sort a vector of reals. By using the polymorphism in Napier we can write one procedure for all types, instead of a different one for each type. This greatly reduces the amount of code that has to be written in a large system.

Experienced Ada users will recognise that the generic facility has similar properties[10]. There are, however, some important differences between generics and the type polymorphism described above and we will return to this later in this paper. For the moment we will develop our argument by the use of an example to construct an index from integers to objects of any type.

First we will write an index from integers to strings. The index is implemented by a binary tree where each leaf contains the integer key and the string value, as well as its left and right subtrees. There are three procedures concerned with maintaining the index, one to create the initial index, one to enter values by key, and one to retrieve values by key. They may be written as in Figure 1.

```
rec type s_index is variant (s_node : S_Node ; tip : null)
& S_Node is structure (key : int ; value : string ; left, right : s_index)

let s_create = proc (-> s_index) ; s_index (tip : nil)
        !Return the empty index by injecting the nil value into the variant

rec let s_enter = proc (k : int ; v : string ; i : s_index -> s_index)
        !Enter the value into the binary tree indexed by key
if i is tip then s_index (s_node : S_Node (k, v, s_create (), s_create ())) else
case true of
        k < i's_node (key)  : { i's_node (left) := s_enter (k, v, i's_node (left)) ; i }
        k > i's_node (key)  : { i's_node (right) := s_enter (k, v, i's_node (right)) ; i }
default                     : { i's_node (value) := v ; i }

let s_lookup = proc (k : int ; fail_value : string ; i : s_index -> string)
        !lookup the value in the binary tree
begin
        let head := i
        while head is s_node and k ≠ head's_node (key) do
        head := if k < head's_node (key)  then head's_node (left)
                                          else head's_node (right)
        if head is s_node then head's_node (value) else fail_value
end
```

Figure 1 An integer to string index

For convenience we will package the two procedures, *enter* and *lookup*, into a structure together with an index that they can operate on. The type of this package is

```
type s_index_pack is structure (
                        Ind     : s_index,
                        Enter   : proc (int, string, s_index -> s_index),
                        Lookup  : proc (int, s_index -> string) )
```

We can now write a procedure to generate different instances of this package. That is the same procedures, *s_enter* and *s_lookup*, but with different indexes. This generating procedure can be written as

```
let s_index_pack_generator = proc ( -> s_index_pack)
                        s_index_pack (s_create (), s_enter, s_lookup)
                        !s_create is called to provide the initial value
```

The generator may now be used to create specific instances of the index. For example

```
let X = s_index_pack_generator ()
```

Thus *X* is a structure with three fields. One is of type *s_index* and the other two are procedures. We can use this structure by, for example

```
X (Ind) := X (Enter)(5, "ron", X (Ind) )
```

This uses the enter procedure in *X* to associate the value "ron" with the key 5 and to overwrite the old index with the new value. We will see later that this generator procedure may be stored away for subsequent reuse in the persistent store.

The first improvement, in the context of software reuse, over this solution is to generalise the type of the stored value. That is, instead of just being an index of strings we wish to have a

pack that will give us an integer index to any type. We will begin, in Figure 2, by making the procedures polymorphic.

```
rec type index [t] is variant (node : Node [t] ; tip : null)
& Node [s] is structure (key : int ; value : s ; left, right : index [s])

let create = proc [t] (-> index [t] ) ; index [t] (tip : nil)
        !Return the empty index by injecting the nil value into the variant

rec let enter = proc [t] (k : int ; v : t ; i : index [t] -> index [t])
!Enter the value into the binary tree indexed by key
if i is tip then index [t] (node : Node [t] (k, v, create [t] (), create [t] ())) else
case true of
        k < i'node (key)  : { i'node (left) := enter [t] (k, v, i'node (left)) ; i }
        k > i'node (key)  : { i'node (right) := enter [t] (k, v, i'node (right)) ; i }
default             : { i'node (value) := v ; i }

let lookup = proc [t] (k : int ; fail_value : t ; i : index [t] -> t)
        !lookup the value in the binary tree
begin
        let head := i
        while head is node and k ≠ head'node (key) do
        head := if k < head'node (key)      then head'node (left)
                                            else head'node (right)
        if head is node then head'node (value) else fail_value
end
```

Figure 2 An integer to any type index

Thus each of the procedures will now act on binary trees of any type. We will again write a generator procedure to produce indexes of a specific type. The type of the new pack is a parameterised type.

```
type index_pack [t] is structure (
                        Ind : index [t],
                        Enter : proc (int, t, index [t] -> index [t]),
                        Lookup : proc (int, index [t] -> t ))
```

The generating procedure can be written as

```
let index_pack_generator = proc [t] ( -> index_pack [t])
                        index_pack [t] (create [t] (), enter [t], lookup [t])
```

The generator may now be used to create specific instances of the index. For example

```
let X = index_pack_generator [string] ()
X (Ind) := X (Enter)(5, "ron", X (Ind) )
```

Another index of integers to pictures could be generated by

```
let Y = index_pack_generator [pic] ()
```

Thus, this polymorphic generating procedure may be used to generate many instances of the index which can vary by type. Again this procedure may be stored for later reuse in the

persistent store but before we describe how that is done we will turn our attention to abstract data types.

2.2 Abstract Data Types

There is a second type of abstraction that we require over our indexes. We may, for example, wish to change the representation of the index and then construct programs that will work for indexes of all representations. To do this consistently we must be able to hide the representation of the index. At present this cannot be done since we use structural equivalence and may discover a type by merely writing it down. We now require the power of abstract data types.

We return to our index of strings. An abstract data type for such an object can be defined in Napier by

```
type abs_s_index is abstype [S_index] (
                    Ind     : S_index ;
                    Enter   : proc (int, string, S_index -> S_index) ;
                    Lookup  : proc (int, string, S_index -> string) )
```

All we know about an object of this type is that it is a structure with fields *Ind*, *Enter* and *Lookup* corresponding to the above description for the same type *S_index*. Outside the abstract data we cannot see the representation of the index and thus we have abstracted over the witness type *S_index*. This type of object is an existentially quantified type[14],

```
    ∃S_index.abs (  Ind     : S_index
                    Enter   : proc (int, string, S_index -> S_index)
                    Lookup  : proc (int, S_index -> string) )
```

That is, we know that such a type *S_index* exists but nothing more. Once we have created an object of type abs_s_index we can no longer tell anything about the representation used for *S_index*. We can create such an object by

```
let abs_S_index = abs_s_index [s_index] (s_create (), s_enter, s_lookup)
```

Here we have used the procedures *s_create*, *s_ enter* and *s_lookup* with the type *s_index* defined earlier in Figure 1. Once we have created this object we can no longer tell that it uses a binary tree to represent the index since it is exactly this information that is abstract in the object. We may have many index objects all with different representations but all with the same abstract type. We can write further procedures to work on this abstract type. For example

```
let abs_s_enter = proc (k : int ; s : string ; i : abs_s_index -> abs_s_index)
begin
    use i as X in
    begin
        X (Ind) := X (Enter) (k, s, X (Ind))
    end
    i
end
```

The **use** clause is a scoping and renaming device. The abstract data type is renamed as *X* in the clause following the **in**. By giving the object a constant name, *X*, we can ensure statically that the interface procedures will only be applied to the object of the correct representation. Indeed it is even stronger than this since objects named by fields can only operate on other fields of the same *X* as they are the only ones that we know are of the same representation, whatever it might be. The procedure, *abs_s_enter* will work with string indexes no matter how they are

implemented, as long as they are of this abstract type, and it is trivial to write one for *lookup*. For example

```
let abs_s_lookup = proc (k : int ; s : string; i : abs_s_index -> string)
begin
        let s1 := ""
        use i as X in s1 := X (Lookup)(k, s, X (Ind))
        s1
end
```

The generating procedure for this abstract data type could be

```
let abs_s_index_generator = proc( -> abs_s_index)
                        abs_s_index [s_index] (s_create (), s_enter, s_lookup)
```

The important point about the procedures *abs_s_enter* and *abs_s_lookup* is that they will operate on objects of this abstract types irrespective of their implementation. Thus, if we had two procedures that operated on B-trees we could package them with an appropriate location and the object would be applicable to the procedures *abs_s_enter* and *abs_s_lookup*. For example

```
type B_tree is variant ( ... )
let B_enter = proc (k : int ; s : string; i : B_tree -> B_tree) ; ...
let B_lookup = proc (k : int ; i : B_tree -> string) ; ...
let abs_B_tree_index = abs_s_index [B_tree] (B_tree (tip : nil), B_enter, B_lookup)
```

abs_B_tree_index is of the same abstract type as *abs_S_index* (i.e. *abs_s_index*) and may be used as a parameter to *abs_s_enter* and *abs_s_lookup*.

We will now combine universal and existential quantification to provide both the abstractions that we require over indexes. The type of a generalised index may be written

```
        type abs.index [t] is abstype [Index] (
                        Ind         : Index ;
                        Enter       : proc (int, t, Index -> Index) ;
                        Lookup      : proc (int, t, Index -> t) )
```

and we can create such an object by a generator procedure. For example

```
        let general_index = proc [t] ( -> abs_index [t])
                                abs_index [t] [index] (create [t](), enter[t], lookup [t])
```

Here we have used the polymorphic procedures *create*, *enter* and *lookup* and the type *index* from Figure 2. *general_index* is different from *index_pack*, which we created earlier in that the procedures, although still quantified by the same type, now make up an abstract data type.

The type of general_index is

```
∀t.proc ( -> ∃Index.abs (   Ind         : Index
                            Enter       : proc (int, t, Index -> Index)
                            Lookup      : proc (int, Index -> t) ) )
```

We can specialise the general index for a particular use. For example

```
let Y = general_index [string] ()
use Y as X in
begin
        X (Ind) := X (Enter) (5, "ron", X (Ind))
        write (X (Lookup) (5, X (Ind)))
end
```

We can, of course, store polymorphic procedures that produce abstract data types and if necessary general procedures to operate on them as before.

The difference in application between universal and existential quantification is that in universal quantification we can write one abstract polymorphic form from which we can generate special cases whereas with existential quantification we describe existing objects by a more general type and thus allow more general abstraction over that type.

Given that we can now program with these objects we will now show how they can be kept in the persistent store.

3. Persistence

We have defined the persistence of data to be the length of time for which data exists and is usable. Thus it is an abstraction over one of the physical properties of data, that of the length of time for which we keep it. Elsewhere[1,3,16] we have described the advantages of not having to explicitly program for the differences in the use of long and short term data and we will not labour them here. It is sufficient to say that by ensuring the persistence abstraction we obtain significant gains in the software engineering of large systems. The figure often quoted is 30% of the total cost of a system throughout its life cycle[1].

In Napier all data is persistent. That is, data is kept for as long as it is usable. This we can determine from the fact that it is reachable by the computation of the transitive closure of objects from the persistence root, called PS. When a program terminates all its data objects may be destroyed except those that the program has arranged to be reachable from PS. It should be noted that in general the persistent store will be a graph and may be distributed over many machines.

The distinguished point in the object graph, PS, has the data type **env** in Napier. Objects of type **env** are collections of bindings, that is, typed, name - value pairs. They differ from structures in that objects of type **env** belong to the infinite union of all such cross products. Furthermore we can add bindings to, or delete bindings from, objects of type **env**.

We will now write program segments to place the data object general_index into the persistent store and another to retrieve it for reuse.

```
let e = environment ()  !create a new empty environment
in e let general_index = proc [t ] ( -> abs_index [t])
                         abs_index [t] [index] (create [t](), enter [t], lookup [t])
                         !Add the general_index to the environment e
in PS let index_env = e
!This new one is the binding index_env : env
```

Figure 3 Binding to an environment

We now have an arrangement where general_index is contained in the environment *index_env* which is itself contained in the environment *PS*. If the program executing this now terminated then *general_index* would be automatically part of the persistent store, since it is reachable from *PS*, and therefore retained.

To retrieve general_index for reuse we could write

```
type abs_index [t] is abstype [Index](
            Ind         : Index
            Enter       : proc (int, t, Index -> Index)
            Lookup      : proc (int, Index -> t) )

use PS as index_env : env in
      use index_env as general_index : proc[t] ( -> abs_index [t] ) in
      begin
            let Y = general_index [string] ()
            use Y as X in
                  begin
                        X (Ind) := X (Enter) (5, "ron", X (Ind))
                        write (X (Lookup) (5, X (Ind)))
                  end
      end
```

Figure 4 Reusing objects in an environment

In this example, there are two distinct uses of the **use** clause. We have seen the one to bind abstract data types to constant names before. The second method binds an environment and some of its field names to the clause following **in**. For example

use PS **as** index_env : **env in** ...

allows us to use the name index_env with type **env** in the clause following **in**. The binding which occurs at run time, and is therefore dynamic, is similar to projecting out of a union. The difference here is that we only require a partial match on the fields. Other fields not mentioned in the **use** are invisible in the qualified clause and may not be used.

This method of dynamic composition or binding to environments allows us to compose systems while still retaining static type checking. It should be noticed that one of the advantages of structural equivalence of type is that two types in different programs may be the same and therefore we can determine type equivalence across program boundaries.

The environment mechanism in Napier also provides a contextual naming scheme. The main mechanism for controlling the naming of objects is scope. This can be a very powerful device especially in languages with higher order procedures[3]. The **use** clause introduces names into a context in the same manner as declarations in blocks. Blocks, however, are nested statically whereas the environments can be composed at run time and therefore nested dynamically. This gives a method of binding commonly found in operating systems and suggests that it is best used in that type of activity. That is, when we want to compose objects out of already existing components.

We have chosen this mixture of static and dynamic checking schemes to preserve the inherent simplicity, safety and efficiency of static checking without insisting that the whole system be statically bound. The cost of total static binding in an object system is that alteration to any part of the schema involves total recompilation of the whole system. This is usually an unacceptably high cost in most systems.

The cost of total dynamic checking is that it is harder to reason about programs statically, it is less safe in that errors occur later in the life cycle and perhaps at dangerous moments and that it is less efficient in terms of cost since errors appear later and also in machine efficiency since we cannot factor out static information.

A judicious mixture of static and dynamic checking is therefore necessary to avoid either of the above extremes, and we propose the above where dynamic checking is only required on projection from a union.

Other approaches to solving the problems of software reuse in a manner similar to the above can be found in SMALLTALK [9], which used inheritence polymorphism and dynamic type checking; Pebble [6] with dependent types in the value space; and the Abstract Data Store [18]

with persistence and dynamic type checking. All three approaches utilise more dynamic binding than Napier.

4. Conclusions

We have described, by example, how a powerful polymorphic type system may be used in the context of software reuse. A type system, for the language Napier, is described with emphasis on universal and existential types and the type **env**. Universally quantified types allow objects to be abstracted over by type. This is similar to and has the same advantages as generics in Ada. The difference is that Ada requires a compiler and a library system to control the use of generics. There are no objects of type generic at run time in Ada.

Since the persistent store in Napier subsumes the need for a library, and indeed most of the functions of an APSE, we can rely on one mechanism, the type checker, to ensure the correct use of generics.

The type system of Napier also allows for abstract data types. Ada again has a similar notion in packages with limited private types. The main difference here is that since packages are not part of the Ada type system, we cannot parameterise one package by another. This is a simple matter in Napier. Furthermore neither generics nor packages may be part of a program data structure.

The search for a richer and more powerful type system is not sufficient to justify the end. In a persistent system the advantage of such a type structure is that we can use one mechanism, instead of a plethora, for checking the legal composition of objects. Thus the system is simpler to use with all the attendant benefits for software costs.

We have included in the type system of Napier the infinite union of typed, name-value bindings. This yields two advantages. The first is that since it is a union, projection out of the union can only be performed at run time, thus yielding a dynamic binding. The second advantage is gained by combining this dynamic check with names to yield a dynamic name composition technique.

Our skill in component reuse will be in deciding which components are complete and may be statically frozen and which components are better dynamically composed.

5. Acknowledgements

We acknowledge the many discussions with our collaborators on the PISA project. In particular, John Scott of STC and Mike Livesey for their many insights into type systems. The work is supported by SERC grants GR/D 4326.6 and GR/D and a grant from STC.

6. References

1. Atkinson,M.P., Bailey,P.J., Chisholm,K.J., Cockshott,W.P. & Morrison,R. "An approach to persistent programming". Computer Journal 26,4 (November 1983),360-365.

2. Atkinson, M.P. & Morrison, R. "Types, bindings and parameters in a persistent environment". Proc of the Appin Workshop on Data Types and Persistence, Universities of Glasgow and St Andrews, PPRR-16, (August 1985),1-25

3. Atkinson, M.P. & Morrison, R. "Procedures as persistent data objects". ACM.TOPLAS 7,4 (October 1985),539-559.

4. Atkinson, M.P., Morrison, R & Pratten, G.. "Designing a persistent information space architecture". 10th IFIP World Congress, Dublin (September 1986),115-120. North-Holland, Amsterdam.

5. Boehm, B.W. "Understanding and controlling software costs". 10th IFIP World Congress, Dublin (September 1986), 703-714. North-Holland, Amsterdam.

6. Burstal, R. & Lampson, B. "A kernal language for abstract data types and modules". Proc. international symposium on the semantics of data types, Sophia-Antipolis, France (1984). In **Lecture Notes in Computer Science**. 173 Springer-Verlag (1984).

7. Cardelli, L. & Wegner, P. "On understanding types, data abstraction and polymorphism". ACM.Computing Surveys 17, 4 (December 1985), 471-523.

8. Demers, A. & Donahue, J. "Revised report on Russell". Technical report TR79-389, (1979), Cornell University.

9. Goldberg, A. & Robson, D. **SMALLTALK-80 The language and its implementation**. Addison-Wesley, London. (1983).

10. Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).

11. Matthews, D.C.J. Poly manual. Technical Report 65 (1985), University of Cambridge, U.K.

12. Milner, R. "A theory of type polymorphism in programming". JACM 26(4), 792-818.

13. Milner, R. "A proposal for standard ML". Technical Report CSR-157-83. University of Edinburgh.

14. Mitchell, J.C. & Plotkin, G.D "Abstract types have existential type". Proc POPL 1985.

15. Morrison, R., Bailey, P.J., Brown, A.L., Dearle, A. & Atkinson, M.P. "A persistent store as an enabling technology for an integrated project support environment. IEEE 8th International Conference on Software Engineering, London (August 1985),166-172.

16. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. "An integrated graphics programming environment". 4th UK Eurographics Conference, Glasgow (March 1986). In Computer Graphics Forum 5, 2 (June 1986),147-157.

17. Morrison, R., Brown, A.L.,Carrick, R. Conner, R.C. & Dearle, A Napier language reference manual. in preparation

18. Powell, M.S. "Adding programming facilities to an abstract data store". Proc of the Appin Workshop on Data Types and Persistence, Universities of Glasgow and St Andrews, PPRR-16, (August 1985),139-160.