The Napier88 Persistent Programming Language and Environment

Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Munro, D.S. & Atkinson, M.P.[§]

School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland

[§]Department of Computer Science, University of Glasgow, Glasgow G12 8QQ, Scotland

Abstract. Persistent programming systems are designed as an implementation technology for long lived, concurrently accessed and potentially large bodies of data and programs, known here as persistent application systems (PASs). Within a PAS the persistence concept is 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. As such it is seen as having a number of benefits in simplifying the task of PAS programming. Here, we describe the integrated design of the Napier88 persistent programming system and how Napier88 may be used to develop PASs.

1 Introduction

The Napier88 persistent programming system was originally planned as part of the PISA project [1] with the major goal of constructing a self contained, orthogonally persistent system. The system was also intended as, or turned out to be, a testbed for experiments in: type systems for data modelling [2-7], bulk data [8, 9] and protection [10, 11]; programming language implementation [12, 13]; binding mechanisms [14-17]; programming environments [17-20]; system evolution [21-23]; concurrency control and transactions [24-27]; object stores [26, 28-35] and software engineering tools [36-39].

The Napier88 system consists of the Napier88 language [40] and its persistent environment [41]. The persistent store comes pre-populated, rather like the SMALLTALK Virtual Image [42], and indeed the system uses values within the persistent store to support itself. The programmer is able to operate entirely within the persistent environment which provides editors, window managers, compilers etc.

Unlike its predecessor, PS-algol [43], which took the approach of extending an existing programming language, S-algol [44], with persistence, Napier88 was designed as an integrated persistent programming system [45]. As such some of the decisions as to what is built into the language and what is supported by the environment are somewhat arbitrary and would justify re-evaluation in future designs. For example, bulk types and concurrency control are supported by values and procedures in the environment whereas graphics facilities, both raster [46] and line drawing [47], are supported in the base language. In particular, there was no attempt to define a minimal language with support facilities being supplied by the persistent environment; rather the design attempted to separate the concepts required for persistent programming and to provide a powerful composition mechanism that allowed these concepts to be freely combined.

The Napier88 system is designed as a layered architecture [34] as shown in Figure 1. All the architectural layers are virtual in that, in any implementation, they may be implemented separately or together as efficiency dictates. Thus, they are definitional rather than concrete.

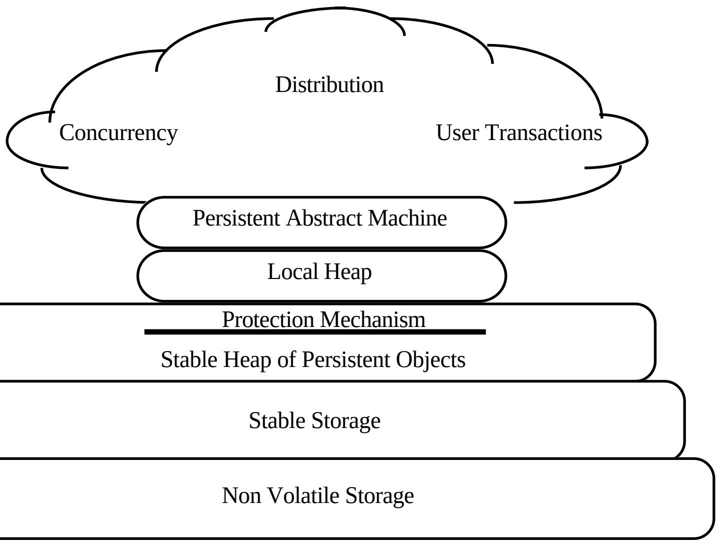


Fig. 1. The Napier88 layered architecture

The success of the Napier88 architecture is highlighted by the fact that it was also used in the implementations of Galileo [48], P-Quest [49] and Staple [50].

The Flask architecture [26] was developed from the above to accommodate concurrency control and distribution [51]. This architecture is shown in Figure 2 as a "V-shaped" layered architecture to signify the minimal functionality built-in at the lower layers. At the top layer the specifications of the model are independent of the algorithms used to enforce them and can take advantage of the semantics of these algorithms to exploit potential concurrency [52]. The information gleaned from the specifications is fed down to the lower layers to aid their efficiency. More importantly such an approach is flexible enough to accommodate different models of concurrency control and distribution. This is just one example of how the closed persistent world can use its high-level semantics to guide the execution of systems.

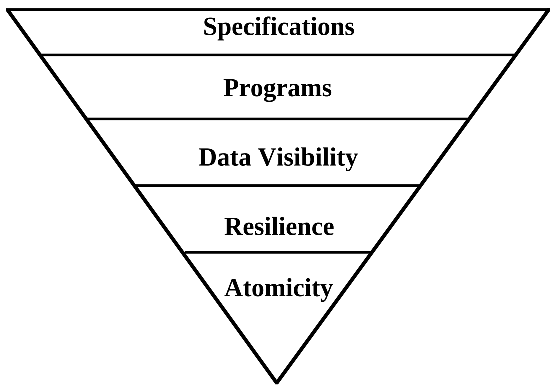


Fig. 2. The Flask Architecture

A large body of technology such as compilers, browsers, object stores, abstract machines, garbage collectors and hyper-programming facilities underlies the implementation of the Napier88 system. These are described elsewhere. Here we concentrate of the design philosophy of the system and the techniques for persistent programming. This is done through the following areas: controlling complexity, the provision of orthogonal persistence, data modelling, the protection of data, controlled system evolution, concurrency control and transactions, and programming within the persistent environment, including hyper-programming.

The justification of the persistence design decisions is given in [53] and the advantages of the abstraction outlined in [4, 11, 15, 18, 34, 54-63].

2 Controlling Complexity

2.1 Language Design

McCarthy [64], van Wijngaarden [65], Strachey [66] and Tennent [67] all observed that expressive power in programming languages could be gained by separating the underlying concepts and allowing them to be combined by powerful composition rules. Strachey and later Tennent distilled these ideas into three principles for use in the design of programming languages:

- the principle of correspondence,
- the principle of abstraction and
- the principle of data type completeness.

The principle of correspondence states that the rules governing the use of names and bindings in a programming language should be consistent. In particular the rules for introducing names and bindings in declarations should have a corresponding mechanism for abstraction parameters. This ensures that formal parameters behave consistently with local declarations. In Napier88 there is only one parameter passing technique, call by value, and there is a 1-1 correspondence between declarations and parameter passing modes.

The principle of abstraction states that for all significant syntactic categories in the language there should be an abstraction mechanism. This allows essential details to be ignored by concentrating on the general structure. An abstraction consists of naming the syntactic category and allowing it to be parameterised. The Napier88 forms of abstractions are procedures and abstract data types in the value space and parameterised types in the type space.

The principle of data type completeness states that any combination or construction of data should be allowed for all types. As a consequence all data objects in a language should have the same "civil rights". More of this shortly.

The overall goal of the above principles is to design languages that are both simple and powerful. They are simple in that there are a minimum of defining rules with no exceptions, since for every exception to a rule the language becomes more complicated in terms of understandability and implementation. The minimisation of defining rules without exceptions also contributes to the power of the language since every exception makes the language less powerful in that it introduces a restriction. The expressive power therefore comes from ensuring that the composition rules are complete and minimal with no exceptions.

As an example of the application of the above principles, we highlight the design of the Napier88 type system which defines the Universe of Discourse of the language.

The Napier88 type system is based on the notion of types as a set structure imposed over the value space. Membership of the type sets is defined in terms of common attributes possessed by values, such as the operations defined over them. In the absence of polymorphism these sets or types partition the value space; polymorphic forms, which in Napier88 are polymorphic procedures and abstract data types, allow values to belong to more than a single type [68]. The sets may be predefined, like *integer*, or they may be formed by using one of the predefined type constructors, like *structure*. The Universe of Discourse of Napier88 is defined by the following rules and is discussed later in the section on data modelling:

1. The scalar data types are *int*, *real*, *bool*, *pixel*, *file* and *null*.

2. Type *string* is the type of a sequence of characters; this type embraces the empty string and single characters.

3. Type *pic* is the type of a conceptual line drawing, modelled in an infinite 2-D real space; this type embraces single points.

4. Type *image* is the type of a value consisting of a rectangular matrix of pixels.

5. Type *env* is the type of an environment; a value of this type consists of a collection of bindings.

6. Type *any* is an infinite union type; a value of this type consists of a value of any type together with a representation of that type.

The following type constructors are defined in Napier88:

7. For any type t, $*t$ is the type of a vector with elements of type t.

8. For labels $I_1,...,I_n$ 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 and $n \geq 0$.

9. For labels $I_1,...,I_n$ and types $t_1,...,t_n$, *variant* $(I_1: t_1,...,I_n: t_n)$ is the type of a variant with labels I_i and corresponding types t_i, for i = 1..n and $n \geq 0$.

10. For any types $t_1,...,t_n$ and t, *proc* $(t_1,...,t_n \rightarrow t)$ is the type of a procedure with parameter types t_i, for i = 1..n, where $n \geq 0$, and result type t. The type of a resultless procedure is *proc* $(t_1,...,t_n)$.

11. *proc* $[T_1,...,T_m]$ $(t_1,...,t_n \rightarrow t)$, where the definitions of types $t_1,...,t_n$ and t may include the use of the type variables $T_1,...,T_m$, is the type of a procedure which is universally quantified over these type variables for $m > 0$ and $n \geq 0$. These are polymorphic procedures. The type of a resultless polymorphic procedure is *proc* $[T_1,...,T_m]$ $(t_1,...,t_n)$.

12. *abstype* $[W_1,...,W_m]$ $(I_1: t_1,...,I_n: t_n)$, where the definitions of types $t_1,...,t_n$ may include the use of the type variables $W_1,...,W_m$, is the type of a structure which is existentially quantified over these type variables for $m > 0$ and $n \geq 0$. These are abstract data types.

The world of data values is defined by the closure of rules 1 to 6 under the recursive application of rules 7 to 12.

An essential element for controlling complexity is that there should be a high degree of abstraction. Thus, in the above type rules, 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 and type parameterisation as abstractions over type. The infinite unions **env** and **any** are used to support persistence, as well as being a general modelling technique; they are dynamically checked.

The type constructors of Napier88 obey the *Principle of Data Type Completeness*, in that, where a type may be used in a constructor, any type is legal without exception. Thus all data values are first class.

2.2 Orthogonal Persistence

The implication of orthogonal persistence is that the user need never write code to move or convert data for long or short term storage [53]. There are three design principles that may be used to achieve orthogonal persistence. They are:

The Principle of Persistence Independence

> The form of a program is independent of the longevity of the data that it manipulates. Programs look the same whether they manipulate short-term or long-term data.

The Principle of Data Type Orthogonality

> All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.

The Principle of Persistence Identification

> The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system.

The application of these three principles yields ***Orthogonal Persistence***.

Persistence independence frees the programmer from the burden of having to explicitly program the movement of data among the hierarchy of storage devices and from coding translations between long-term and short-term representations; this is performed automatically by the system. The mechanical cost of performing the movement of data does not disappear but the intellectual cost does. That is, the programmer need not specifically write code for it, making the application code smaller and more intellectually manageable. However, the implementor of the support system now has the challenge of automating that data movement and any translation efficiently.

Data type orthogonality is an aid to data modelling in that it ensures that the data model can be complete and independent of the persistence of the data being modelled. For example, bulk data types abstract over size and are therefore commonly used in persistent programming languages to aid the manipulation of massive collections of data such as scanned data from satellites or insurance policies sold by a company. Where such data is only considered long-term then the data model has to allow explicit conversion between long and short-term forms to facilitate creation of new bulk data and the manipulation of extracts from the long-term bulk data as short-term data.

Persistence identification may be satisfied by identifying the persistent data automatically. In a natural extension of the garbage collection technique, where only useful data survives, the persistent data of a program can be found by computing the transitive closure of all the data from a number of distinguished roots. This technique is called *identification by reachability* and is the one now most commonly used in orthogonally persistent systems. Such conversions are a distraction when the PAS programmer has to code them.

The Napier88 system obeys all three of the above principles of persistence. Programs operate on long and short term data identically, all data types may persist and the useful data is identified by reachability.

3 Using Persistent Data

Napier88 programs may access their home persistent environment by calling the only pre-defined procedure in the language, *PS*. Other persistent stores may also be accessed and values from those stores copied into the home environment. Each persistent store is organised as a graph of objects but the topology of the graphs may vary from store to store. The *PS* procedure is specified as follows:

```
PS: proc (→ any)
```

That is, *PS* is a procedure that takes no parameters and yields a value, the root of persistence, of type **any**. The root of persistence is constant and may not be altered after the store has been created. However the values reachable from the root may be updated if they are variable.

Type **any** is the infinite union of all types into which values may be injected (coerced) dynamically. Stores are initialised by a standard procedure that takes a parameter of type **any** which will have the initial store value injected into it. The mechanism for creating stores is contained in [41].

To retrieve values from the persistent store, a projection operation (coercion), from type **any**, is required. So, for example, if a persistent store consisted of one value which was a structure with a *name* and *address* in it, both of type **string**, then the following program fragment could be used to retrieve the value:

```
type person is structure (name, address : string)
! This declares a type, person, as a structure (labelled cross product)
! with two fields (name and address) of type string

let ps = PS () ! Calling the PS predefined procedure yields
                        ! the root of persistence.
                        ! It is then declared as the constant ps.
                        ! Constancy is denoted by the =
                        ! variables are declared using :=
project ps as X onto
person        : ! The value X may be used here with type person
default: {} ! This is the catch all and ps has type any here
```

Fig. 3. Using Persistent Data

The **project** clause takes a value, *ps* in this case, and either coerces the type to one in the list of types that follows or executes the **default** option. The value is also given a constant identifier, *X* in this case, to avoid side effects. This identifier is in scope on the right hand side of the colon and has the type on the left hand side. In practice, the persistent stores will be much more complicated, often forming cyclic structures.

The movement of data for execution is hidden from the user. When the program accesses a value it is automatically made available, thus the user may abstract over this physical property of the store. To change values in the persistent store, they must first be accessed and then updated. At the end of the program execution the transitive closure of the values from the original root of persistence is calculated and these values preserved in the persistent store. This action may change the topology of the store but not the original root which is constant once created.

```
type person is structure (name, address : string)

let ps = PS ()

project ps as X onto
person :      begin
      X (name) := "Ronald Morrison”
              X (address) := "St Andrews"
      end
default       : {} ! This is the catch all and ps has type any here
```

Fig. 4. Changing persistent values

In Figure 4, the value that is contained in the persistent store is a structure. Its fields have been updated, using the indexing *X (name)* and *X (address)*, to record the information on a very nice person. At the end of the program, all the information reachable from the original root is preserved. Thus the changes will be preserved until, at least, the next activation of the store whereupon they may be changed again.

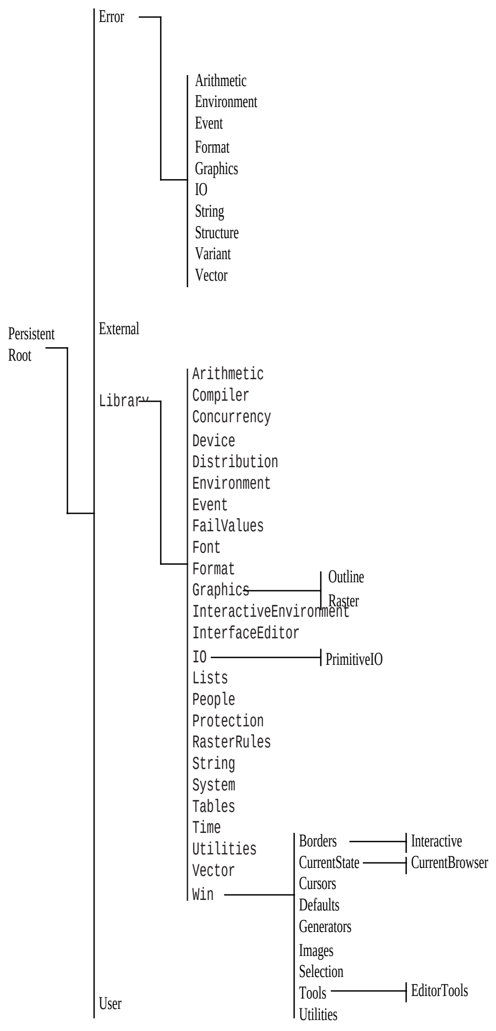


Fig. 5. The Standard release persistent store structure

Figure 5 outlines the initial structure of the Napier88 standard release system 2.0. Most of the names represent environment (**env**) objects which are collections of bindings. The type **env** is the infinite union of all labelled cross products but environments differ from structures in that bindings may be added and removed dynamically. This in turn forces a dynamic type check on the first use of an environment object to ensure that it contains the bindings assumed in the programs. Once this has been established, static type checking is restored.

Figure 6 stores a procedure in the *User* environment for later use. The procedure is part of a complex number package which may be used or added to later. The **in** clause places the value that is declared, *add*, into the environment, *user*. At the end of the program this value will be reachable from the persistent root, via *user*, and is therefore preserved. The binding that is placed in the environment is an *L-value* binding in that it contains a location.

```
let ps = PS ()

project ps as X onto
env   : use X with User : env in
        begin
              type complex is structure (rpart, ipart : real)
              in User let add = proc (a, b : complex → complex)
                        complex (a (rpart) + b (rpart), a (ipart) + b (ipart))
        end
default : {} ! This is executed if the projection fails
```

Fig. 6. Placing a procedure in the persistent store

Figure 7 shows how a second procedure may be placed in the same, *User*, environment. In this way the package can be built up incrementally.

```
let ps = PS ()

project ps as X onto
env   : use X with User : env in
        begin
              type complex is structure (rpart, ipart : real)
              in User let subtract = proc (a, b : complex → complex)
                        complex (a (rpart) - b (rpart), a (ipart) - b (ipart))
        end
default : {} ! This is executed if the projection fails
```

Fig. 7. Adding a procedure to the *User* environment

Figure 8 demonstrates how these procedures may now be accessed in a program. Notice that the values *one*, *two*, *three*, and *minusOne* are not preserved when the program terminates since they are not reachable from the root of persistence.

```
type complex is structure (rpart, ipart : real)

let ps = PS ()

project ps as X onto
env :   use X with User : env in
        use User with add, subtract : proc (a, b : complex → complex) in
        begin
                let one = complex (1.0, 1.0) ; let two = complex (2.0, 2.0)
                let three = add (one, two)
                let minusOne = subtract (one, two)
        end
default : {} ! This is executed if the projection fails
```

Fig. 8. Using the procedures in the *User* environment

The second **use** clause projects two procedure values, *add* and *subtract*, into the current scope. The semantics of the clause is the same as declaring these values within the block except that they are reachable from the root of persistence.

The need for dynamic type checking should now be apparent. The dynamic check is required to ensure that the given environment has the values of the correct type. If it does then the values are placed in scope and may be used like any other value. Otherwise an error condition is raised. All subsequent type checking is static since the compiler can compile the code against the static assumption that the dynamic check will work.

The environment mechanism [69] provides a contextual naming scheme that can be composed dynamically. All values in Napier88 are anonymous. However, values may be contained in more than one environment with different names. This allows different name spaces to be placed over the value space enabling applications to utilise their private persistent name space while sharing values in the persistent store.

The syntactic noise involved in using the persistent store, through **project** and **use** clauses, may be reduced by hyper-programming [20] which allows values to be directly linked to source code at composition time. Thus the navigational code and type specifications need not always be included in the program.

4 Data Modelling

Type systems provide two important facilities within both databases and programming languages, namely data modelling and data protection. Data modelling is performed in databases using data models, which have types to describe the form of the data, and in programming languages by using a classical type system. In both cases the universe of discourse of the system is defined by the set of allowable types which in turn are denoted by the set of legal expressions in the language. Data protection is provided by enforcing explicit and implicit integrity constraints in databases and by type checking in programming languages. A goal of persistent programming language design is to develop a type system that will accommodate the structures required for both modelling and protection in less traditional database applications such as scientific programming, engineering applications and office automation, whilst also capturing the type description of more conventional database systems [70].

As a first step in the unification of data models and type systems some *approximate* equivalences can be recognised. These are summarised in Table 1.

Table 1. Equivalences between data models and type systems

Databases	**Programming Languages**
data models	type systems
schema	type expression
database	variable
database extent	value

The issue of type checking is central to a type system that will provide data modelling and protection for persistent systems. Generally, data models in databases are concerned with the manipulation of the data that is consistent with the constraints imposed by the data model. In some cases these constraints may depend upon values calculated during the computation. As such they can be dynamic in nature and require dynamic integrity constraint checking for enforcement. By contrast classical type systems for programming languages are concerned with static checking which allows assertions to be made and even proved about a computation before it is executed. Static checking therefore provides a level of safety within the system. It also allows more efficient code since type checking code is not required at run-time.

At first the dichotomy between the checking times in databases and programming languages appears to be beyond resolution. The Napier88 approach is to ensure that type system is mostly statically checkable. However, some dynamic checking on projection out of unions for types **any** and **env**, as well as variant selection, allows the dynamic binding required for orthogonal persistence [53] and system evolution [23].

For data modelling, Napier88 provides the base types integer, real, boolean, string, pixel and the trivial type null. The type system also includes graphical types for line drawing in an infinite two-dimensional real space and for manipulating raster images which are aggregates of pixels. The general aggregates are structure (labelled cross product, x) and vector. Variants are available for labelled disjoint sums (+). For abstraction the type system provides procedures and abstract data types. The Napier88 type system is polymorphic, like ML [71, 72], Russell [73] and Poly [74] and uses the existentially quantified types of Mitchell & Plotkin [5, 75] for abstract data types.

There is deliberately no type inference, to allow for explicit specialisation of polymorphic forms from the persistent store. Thus polymorphic forms may be stored in the persistent store and, when accessed, specialised with their specialising type.

A unique design feature of the implementation of the typed objects is that their storage format may be non-uniform [12].

The type equivalence rule in Napier88 is by structure to allow separately prepared programs and data to be composed without reference to a common scheme. To accommodate expressive power, type parameterisation is provided and both recursive and parameterised types are allowed in the type algebra. This combination with structural equivalence, in general, leads to undecidable type checking. The solution in Napier88 is a syntactic convention which allows the type checking to be sound, complete and co-complete [4].

4.1 User Defined Types

Napier88 provides the user with the ability to assign names to type constructions. For example,

```
type complex is structure (rpart, ipart : real)
```

declares a structure (labelled cross product) with two fields *rpart* and *ipart* both of type real. The structure type is given the local name *complex* which may now be used as a shorthand for the type expression.

Recursive types provide the ability to define cyclic data types. A binary tree can be represented by a type definition in which each node has a *key* field of type **int** and a value field of type **string**. Each node of the binary tree also has a *left* and a *right* field of the tree type, *intStrBTree*, to point to the descendent sub-trees. The nodes of the tree are represented by a structure type and the tree itself, with all its sub-trees, can be either a node or an empty tree represented by the **null** type. The following declaration of the type name, *intStrBTree*, defines such a binary tree.

```
rec type intStrBTree is variant (node : Node ; tip : null)
& Node is structure (key : int ; value : string ; left, right : intStrBTree)
```

Expressive power is further increased by type operators which allow types to be parameterised by others. For example

```
type Pair [t] is structure (first, second : t)
```

Pair [t] is not strictly a type but a type operator which defines an infinite class of types. It may be parameterised by a type to produce a specific type definition. For example:

```
type intPair is Pair [int]            ! This is equivalent to
                                         ! structure (first, second : int)
type imagePair is Pair [image]  ! This is equivalent to
                                         ! structure (first, second : image)
```

The combination of recursive definition and type parameterisation yields further expressive power. For example the following type operator defines an infinite class of binary trees.

```
rec type binaryTree [Key, Value] is variant (   node : Node [Key, Value] ;
                                                   tip : null)
& Node [Key, Value] is structure (   key : Key ; value : Value ;
                                        left, right : binaryTree [Key, Value] )
```

These may be used as before to define specific binary trees.

```
type intStringBinaryTree is binaryTree [int, string]

type stringImageBinaryTree is binaryTree [string, image]
```

The uncontrolled introduction of recursive type operators leads to the ability to describe types over which no decidable structural equivalence algorithm is known. Napier88 therefore restricts these definitions by the following rule in order to retain decidable type checking [3]:

> The specialisation of a recursive operator on the right hand side of its own definition may not include any types which are constructed over its own formal parameters.

The importance of type declarations is that they allow the programmer to introduce new and succinct notations and to assign them names that are meaningful within the application being constructed. It therefore aids the traditional role of Data Description Languages by allowing data to be accurately described.

4.2 Polymorphism

In [76] and its companion papers, an analysis of what constitutes a persistent type system is given. For modelling purposes it is generally agreed that some form of polymorphism is required to capture the expressiveness of data models and to increase component re-use [18]. The most favoured forms of polymorphism are universal polymorphism: parametric or inclusion.

Parametric polymorphism describes the polymorphism found in ML [71] and its derivatives whereas inclusion polymorphism is the style of polymorphism found in object-oriented languages such as Simula67 [77]. An interesting hybrid may be found in the database programming language Galileo [48], which is a derivative of ML but utilises inclusion polymorphism to implement part of the Semantic Data Model [78]. Cardelli [68] has shown separately how the parametric and inclusion forms of polymorphism may be integrated, as bounded quantification, to yield forms of abstraction not available to either one.

Napier88 provides parametric polymorphism in the form of universally quantified procedures and existentially quantified types (abstract data types). The utility of these mechanisms will be illustrated with the aid of an example that implements an index. The index will initially use an integer key and store a string value. Figure 9 defines a procedure that generates the index and places the index together with a procedure to enter values into the index and another procedure to lookup values in the index, in an environment. The generating procedure takes as parameters a fail value and an environment in which to place the index and the two procedures. The fail value will be returned by the lookup procedure if the key is not valid. The index is implemented as a binary tree.

```
let generateIntStringIndex = proc (failValue : string ; envir : env)
begin
    rec type index is variant (node : Node ; tip : null)
    & Node is structure (key : int ; value : string ; left, right : index)

    let nullIndex = index (tip : nil)
    ! Construct the empty index by injecting the nil value into the variant

    in envir let i := nullIndex
    ! This is the internal index structure initialisation

    in envir rec let enter = proc (k : int ; v : string ; i : index → index)
    ! Enter the value into the binary tree indexed by key 'k'
    if i is tip then index (node : Node (k, v, nullIndex, nullIndex)) else
    case true of
    k < i'node (key)        : {i'node (left) := enter (k, v, i'node (left)) ; i}
    k = i'node (key)        : i                    ! do nothing
    default                 : {i'node (right) := enter (k, v, i'node (right)) ; i}

    in envir let lookup = proc (k : int ; i : index → string)
    !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 failValue
    end
end
```

Fig. 9. The procedure to generate the index

Placing the index in the persistent store involves creating a new environment and making that reachable from the root of persistence. Creating the new environment entails using the standard procedure to create environments. This is kept in the *Library* environment in the persistent store and must be located and called. The new environment is passed to the procedure to generate the index and then placed in the *User* environment with the name *intStringIndex*. All of this is contained in Figure 10.

```
let ps = PS ()

project ps as X onto
env : use X with User, Library : env in
    use Library with Environment : env in
    use Environment with environment : proc (→ env) in
    begin
        let new = environment ()
        generateIntStringIndex ("This is a failure", new)

        in User let intStringIndex = new
end
default : {} ! This is executed if the projection fails
```

Fig. 10. Place the index in the persistent store

Figure 11 demonstrates how the index may now be used from the persistent store. Notice that the new entry that is entered into the index will be preserved in the persistent store after the program terminates since it is reachable from the root of persistence.

```
let ps = PS ()

rec type index is variant (node : Node ; tip : null)
& Node is structure (key : int ; value : string ; left, right : index)

project ps as X onto
env :   use X with User : env in
        use User with intStringIndex : env in
        use intStringIndex with i : index;
                                enter : proc (int, string, index → index);
                                lookup : proc (int, index → string) in
        begin
            i := enter (49, "Ron", i)
        end
default : {} ! This is executed if the projection fails
```

Fig. 11. Using the index

Thus an index from integers to strings has been created and used. If a second index, say from strings to integers, is required a new set of programs has to be written to ensure type compatibility. Alternatively the power of polymorphism can be used to define a generic procedure that will produce the correct index on application. The polymorphic procedure abstracts over the types of the index key and the value but requires an extra parameter to provide an ordering on the keys. This extra parameter, *lessThan*, is a procedure that compares two keys. Figure 12 defines the polymorphic procedure *generateGeneralIndex* which will generate a given index on application. Note that equality is defined over all types in Napier88.

```
let generateGeneralIndex = proc [Key, Value] (
                                   lessThan : proc (Key, Key → bool);
                                   failValue : Value;
                                   envir : env)
begin
   rec type index is variant (node : Node ; tip : null)
   & Node is structure (key : Key ; value : Value ; left, right : index)

   let nullIndex = index (tip : nil)
   ! Construct the empty index by injecting the nil value into the variant

   in envir let i := nullIndex
   ! This is the internal index structure initialisation

   in envir rec let enter = proc (k : Key ; v : Value ; i : index → index)
   ! Enter the value into the binary tree indexed by key 'k'
   if i is tip then index (node : Node (k, v, nullIndex, nullIndex)) else
   case true of
   lessThan (k, i'node (key))
                       : {i'node (left) := enter (k, v, i'node (left)) ; i}
   k = i'node (key)    : i
   default             : {i'node (right) := enter (k, v, i'node (right)) ; i}

   in envir let lookup = proc (k : Key ; i : index → Value)
   ! lookup the value in the binary tree
   begin
      let head := i
      while head is node and k ≠ head'node (key) do
            head :=  if lessThan (k, head'node (key))
                     then head'node (left) else head'node (right)
      if head is node then head'node (value) else failValue
   end
end
```

Fig. 12. A polymorphic generation procedure

It should be noticed how little the polymorphic code changes from the monomorphic form. This means that there is little extra cost in producing such code. Figure 13 illustrates how the *generateGeneralIndex* procedure may be used to create a specific index.

```
let ps = PS ()

project ps as X onto
env : use X with User, Library : env in
    use Library with Environment : env in
    use Environment with environment : proc (→ env) in
    begin
        let new = environment ()

        let stringLessThan = proc (a, b : string → bool) ; a < b
        generateGeneralIndex [string, int] (stringLessThan , -999, new)

        in User let stringIntIndex = new
end
default : {} ! This is executed if the projection fails
```

Fig. 13. Creating an index from strings to integers

Figure 14 illustrates the final step in using the new index. Notice that a type operator is used to generate the correct type. This is equivalent to the original type defined in the polymorphic procedure.

```
let ps = PS ()

rec type Index [Key, Value] is variant (node : Node [Key, Value] ; tip : null)
& Node [Key, Value] is structure (  key : Key ; value : Value ;
                                     left, right : Index [Key, Value] )
type index is Index [string, int]

project ps as X onto
env :   use X with User : env in
        use User with stringIntIndex : env in
        use stringIntIndex with i : index;
                                 enter : proc (string, int, index → index);
                                 lookup : proc (string, index → int) in
        begin
              i := enter ("Ron", 49, i)
        end
default : {} ! This is executed if the projection fails
```

Fig. 14. Using the string to integer index

The advantage of the polymorphic abstraction should be obvious in the context of software reuse. By using the polymorphism in Napier88, one procedure for all types may be written instead of a different one for each pair of types. This polymorphic generating procedure may then be used to generate many instances of the index which can vary by type. In the above examples the indexes are stored in environments in the persistent store. The generating procedure and all the other code fragments may also be stored for later reuse in the persistent store but such details have been omitted here. The major advantage of the combination of polymorphism and persistence is that it greatly reduces the amount of code that has to be written in a large system.

There is a second type of abstraction that may be required over indexes. In the above examples all of the indexes are implemented by binary trees and the data structure implementing these has to be known for correct use. Further abstraction can be had by hiding the implementation while still allowing the user to construct programs that will work for indexes of all implementations. For this, the power of abstract data types is required.

The implementation of the index may be a binary tree, a B-tree, a B^+-tree or a list etc. The essential element is that they all have the same abstract interface of an index value, a procedure that will take a key, a value and an index as parameters and return an index and another procedure that will take a key and an index as parameters and return a value. Such an interface is defined in Napier88 using abstract data types. For example, the type of all integer to string indexes irrespective of their implementation can be written as:

```
type intStringIndex is abstype [index] (
            Index      : index;
            Enter      : proc (int, string, index → index);
            Lookup     : proc (int, index → string))
```

The witness type, *index*, is hidden to the outside of this interface. This is the implementation type. Thus all values of type *intStringIndex* have the same type, no matter which implementation type is used to construct the abstract data type.

Figure 15 illustrates how both kinds of polymorphic abstraction can be combined. The *generateGeneralAbsBtreeIndex* procedure is polymorphic in the key and value types allowing any ordered type to be used as an index key for any other type. The procedure returns an abstract data type which has the implementation of the index encapsulated within it. For each different implementation a different generating procedure is required.

```
type generalAbsIndex [KEY, VALUE] is abstype [index] (
                Index       : index;
                Enter       : proc (KEY, VALUE, index → index);
                Lookup      : proc (KEY, index → VALUE))

let generateGeneralAbsBtreeIndex = proc [Key, Value] (
                lessThan : proc (Key, Key → bool) ;
                failValue : Value → generalAbsIndex [Key, Value])
begin
    rec type index is variant (node : Node ; tip : null)
    & Node is structure (key : Key ; value : Value ; left, right : index)

    let nullIndex = index (tip : nil)
    ! Construct the empty index by injecting the nil value into the variant

    let i := nullIndex
    ! This is the internal index structure initialisation

    rec let enter = proc (k : Key ; v : Value ; i : index → index)
    ! Enter the value into the binary tree indexed by key 'k'
    if i is tip then index (node : Node (k, v, nullIndex, nullIndex)) else
    case true of
    lessThan (k, i’node (key))
                           : {i'node (left) := enter (k, v, i'node (left)) ; i}
    k = i'node (key)       : i
    default                : {i'node (right) := enter (k, v, i'node (right)) ; i}

    let lookup = proc (k : Key ; i : index → Value)
    ! lookup the value in the binary tree
    begin
        let head := i
        while head is node and k ≠ head'node (key) do
                head :=    if lessThan (k, head'node (key))
                           then head'node (left) else head'node (right)
        if head is node then head'node (value) else failValue
    end

    generalAbsIndex [Key, Value] [index] (i, enter, lookup)
end
```

Fig. 15. Combining universal and existential quantification

Figure 16 illustrates how the abstract data type may be used. Two abstract data types, *this* and *that*, are created, one using the binary tree implementation generator and the other using an unspecified list implementation generator. Notice that while the implementation of the abstract data types are incompatible as are the operations over them, a procedure that will operate over both of them may be constructed.

The **use** clause is a scoping and renaming device. The abstract data value is renamed as *X* in the clause following the **in**. By giving the object a constant name, *X*, the application of the interface procedures can be statically checked. This ensures that the interface procedures will only be applied to objects of the same representation. Indeed the rule 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 are known to be of the same representation, whatever it might be.

```
type intStringAbsIndex is abstype [index] (
                Index       : index;
                Enter       : proc (int, string, index → index);
                Lookup      : proc (int, index → string))

let lessThanInt = proc (a, b → bool) ; a < b

let this = generateGeneralAbsBtreeIndex [int, string] (lessThanInt, "")

let that = generateGeneralAbsListIndex [int, string] (lessThanInt, "")

let updateIntStrIndex = proc (adt : intStringAbsIndex ; k : int ; v : string)
use adt as X in
begin
        X (Index) := X (Enter)(k, v, X (Index))
end

updateIntStrIndex (this, 49, "Ron")
updateIntStrIndex (that, 59, "Malcolm")
```

Fig. 16. Using abstract data types

Again, although not demonstrated here all of this code may be placed in the persistent store for later reuse. The difference in application between universal and existential quantification is that in universal quantification abstract polymorphic form can be written from which special cases can be generated whereas with existential quantification existing objects are described by a more general type thereby allowing more general abstraction over that type.

In summary, the power of the Napier88 type system in the context of data modelling is dependent on the following: the base types and type constructors; the ability to have user defined types that may be parameterised and recursive; and the polymorphism facilities, both universal and existential. These facilities combined with the persistent environment provide for traditional data modelling but can also cater for new applications which require concurrency control, protection and schema evolution within the modelling framework.

5 Protection of Data

Persistent object systems, such as Napier88, support large collections of data that have often been constructed incrementally by a community of users [57]. The data is inherently valuable and requires protection from: system malfunction, such as hardware failure; misuse of common facilities, such as the operating system; and finally from users themselves [11]. Hardware malfunction has little to do with software protection and is best dealt with by techniques such as incremental dumping or stability strategies [79, 80]. Here the focus is on software methods designed for the protection of persistent data. Information hiding is one such software technique in which the access to the data, or the type interface to it, is restricted. By varying the restriction, a variable degree of protection may be obtained.

There are three well-known mechanisms which support information hiding within a strongly typed 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 try to perform undesirable accesses. 1st-order information hiding prevents the protected data from being named by untrusted programs allowing access only through a procedural interface. 2nd-order information hiding is somewhere between these two, allowing access mainly through procedures, but also allowing the protected data to be named. The data is viewed through a mechanism which causes type information loss. This ensures only a limited set of operations may be performed on the hidden data.

Napier88 does not provide subtyping and therefore the focus here is on 1st and 2nd-order information hiding as well as on the use of 2nd-order information hiding to implement traditional database viewing mechanisms.

5.1 1st-Order Information Hiding

1st-order information hiding is achieved by allowing access only to a procedural interface that operates over the hidden data. In Napier88, which has first-class procedure values and block-style scoping, access to the data may be removed simply by its name becoming unavailable (out of scope).

As an introductory example consider a random number generator for which the Napier88 code is given in Figure 17. This is written as a generator procedure, *randomGenerator*, which takes an integer seed, *seed*, and returns a procedure value that will yield a sequence of random numbers when repeatedly called. The returned procedure uses a value, *hiddenValue*, that is encapsulated in the closure of the *randomGenerator* procedure. This value is out of scope at the outermost level but still available for use within the inner procedure. Thus the value is hidden, protected and only available for use through the procedural interface. That means that it can only be manipulated through that interface and not in unintended ways.

```
let randomGenerator = proc (seed : int → proc (→ int))
begin
 let hiddenValue := seed

 proc (→ int)
 begin
       hiddenValue := (519 * hiddenValue) rem 8192
       hiddenValue
 end
end

let random = randomGenerator (2111)
let firstRandomNumber = random ()
let secondRandomNumber = random ()
```

Fig. 17. A random number generator

A more sophisticated example of 1st-order information hiding is that of a bounded buffer into which users may place and obtain messages. The bounded buffer is intended to be used by concurrent threads and therefore access to the buffer is synchronised. Concurrency control in Napier88 will be covered later in this paper and it is sufficient here to use semaphores, which are provided by standard procedures in the environment, for synchronisation.

Figure 18 illustrates how the bounded buffer may be implemented in Napier88. The example is polymorphic in that the generating procedure, *bBGen*, will produce buffers of any type. The implementation of the buffer (a vector), the semaphores and the buffer pointers are hidden in the closure of this generating procedure. The result of the generating procedure is a structure which contains one procedure to obtain a value from the buffer, *get*, and one procedure, *put*, to place a value in the buffer. The generating procedure is placed in the *User* environment in the persistent store.

```
type boundedBuffer [t] is structure (get : proc (→ t) ; put : proc (t))

type Semaphore is structure (wait, signal : proc ())

project PS() as X onto
env : use X with Library, User : env in
        use Library with Concurrency : env in
        use Concurrency with semaphoreGen : proc (int → Semaphore) in
begin
        in User
        let bBGen = proc [t] (bufferSize : int ; initV : t → boundedBuffer [t])
        begin
                let ringBuffer = vector 1 to bufferSize of initV
                let avail = semaphoreGen (bufferSize)
                let mutex = semaphoreGen (1)
                let empty = semaphoreGen (0)
                let getPtr := 1 ; let putPtr := 1

                let Get = proc (→ t)
                begin
                        empty (wait) ()
                        mutex (wait) ()
                        let result := ringBuffer (getPtr)
                        getPtr := getPtr rem bufferSize + 1
                        mutex (signal) ()
                        avail (signal) ()
                        result
                end

                let Put = proc (message : t)
                begin
                        avail (wait) ()
                        mutex (wait) ()
                        ringBuffer (putPtr) := message
                        putPtr := putPtr rem bufferSize + 1
                        mutex (signal) ()
                        empty (signal) ()
                end
                boundedBuffer [t] (Get, Put)
        end
end
default : {}
```

Fig. 18. A synchronised polymorphic ring buffer

Figure 19 demonstrates how the polymorphic bounded buffer generator may be used. First it is identified in the persistent store and then initialised to operate on strings. The buffer is initialised to 500 elements in size, each containing the null string. The procedures in the structure are renamed locally for succinctness. Notice that all the details of the implementation are hidden behind the procedural interface and not even mentioned in this code.

```
type boundedBuffer [t] is structure (get : proc (→ t) ; put : proc (t))

project PS () as X onto
env : use X with User : env in
        use User with bBGen : proc [t] (int, t → boundedBuffer [t]) in
begin
        let thisBuffer = bBGen [string] (500, "")

        let get = thisBuffer (get) ; let put = thisBuffer (put)

        put ("Ron Morrison") ; put ("Richard Connor")

        let first = get ()
        ...
end
default : {}
```

Fig. 19. Using the ring buffer

The bounded buffer is designed to operate in the presence of concurrent access. This has not been shown here as it requires threads which are the subject of a later section.

5.2 2nd-order Information Hiding

2nd-order information hiding differs from 1st-order information hiding in that it does not restrict access to the protected values, but instead abstracts over the type in order to restrict the operations allowed on the values. Thus the protected values may be manipulated by some basic operations, such as assignment and perhaps equality, but their full set of operations are not allowed due to the abstracted type view. This allows the implementation objects themselves to be safely placed in the interface with their abstracted type along with the procedures which manipulate them.

Napier88 provides existentially quantified types (abstract data types) to implement 2nd-order information hiding. To illustrate their utility, we will use the example of a banking system, taken from [10], in which customers have access to their accounts through autoteller machines. The autoteller machines have different styles of access to accounts according to which bank the machine belongs. A customer's own bank may have full access to an account whereas another bank may not access the customer’s account balance, but must know if a withdrawal can be made. The purpose of using 2nd-order information hiding is to allow the autotellers to manipulate the account through its abstract interface without knowing its concrete implementation.

The local autoteller machine is operated through the following abstract interface:

failAc	is the value returned by *getAc* if a password check fails
getAc	is a procedure which takes as input an account number, and a password and provided that the password is correct, returns the account, otherwise it returns the fail value *failAc*
withdraw	is a procedure which removes the amount specified from the account. If there are insufficient funds, the procedure returns **false** otherwise it returns **true**
balance	is a procedure which returns the balance in the account
olimit	is a procedure which returns the account overdraft limit
transfer	is a procedure that transfers an amount from one account to another

This interface is captured in the declaration of the type to represent the local teller.

```
type localTeller is abstype [account](
                failAc      : account;
                getAc: proc (int, string → account);
                withdraw    : proc (int, account → bool);
                balance     : proc (account → int);
                olimit: proc (account → int);
                transfer    : proc( int, account, account))
```

Notice that the hidden type, *account*, which is sometimes referred to as the witness type, is available in the interface of the abstract data type. However all that is known about the type is that it exists, not how it is implemented.

Figure 20 defines a procedure that returns a value of the *localTeller* abstract type. This procedure, *createLocalAutoTeller*, has to define a concrete representation for the account and also to define the procedures that operate over the account. The representation of the account needs to hold the balance, the overdraft limit and the pin number (password) for the account. This is done using a structure type, *account*, with obvious field names. Once the procedures have been defined then the values are made into the abstract data type to hide the implementation details. For simplicity, details of synchronisation for concurrent access are omitted.

```
let createLocalAutoTeller = proc (→ localTeller)
begin
        type account is structure (Balance, Limit : int ; Pin : string)
        let failAc = account (0, 0,"")              ! This is a fail value

        let getAc = proc (accNumber : int ; passwd : string → account)
        ! Look up the account number, check user password and if the
        ! password matches the password in the database return the account,
        ! otherwise return a fail value.
        begin
                let new = lookup (accNumber)
                if new (Pin) = passwd then new else failAc
        end

        let withdraw = proc (debit : int ; ac : account → bool)
        ! Withdraw debit pounds from ac.
        begin
                let result = ac (Balance) - debit
                if result > ac (Limit) and debit > 0 then
                begin
                        ac (Balance) := result
                        true
                end else false
        end

        let balance = proc (ac : account → int) ; ac (Balance)
        ! Return the balance of account ac.

        let olimit = proc (ac : account → int) ; ac (Limit)
        ! Return the credit limit of account ac.

        let transfer = proc (amount : int ; from, to : account)
        if amount > 0 and from (Balance) - amount > from (Limit) do
        begin
                from (Balance) := from (Balance) - amount
                to (Balance) := to (Balance) + amount
        end

        ! Make the abstract data type
        localTeller [account] (failAc, getAc, withdraw, balance, olimit, transfer)
end
```

Fig. 20. Creating the local autoteller abstract data type

The *getAc* procedure uses a data structure, *lookup*, which when indexed by the account number yields the account. For brevity the definition of this data structure has not been defined here but may be implemented by any standard technique such as B-trees, hashing etc. This data structure may also be protected by defining it within the closure of the *createLocalAutoTeller* procedure. This would, however, restrict its use to that procedure and a more general solution would be to encapsulate the data structure in a procedure closure that required a password in order to obtain it. The technique is demonstrated later.

Figure 21 shows how the abstract data type may be used to retire with one million pounds.

```
use createAutoTeller () as X in
begin
        let this = X (getAc) (45, "Ronald")
        if this ≠ X (failAc) then
        begin
                let getMoney = X (withdraw) (1000000, this)
                ! Run off and retire with getMoney
        end else ...
        ...
end
```

Fig. 21. Using the auto teller

The interface also contains a procedure which allows a customer with two accounts to transfer money from one to the other via the local autoteller. Figure 22 illustrates how this might be done.

```
use createAutoTeller () as X in
begin
        let mine = X (getAc) (45, "Ronald")
        let myOther = X (getAc) ( 46, "Ann")
        if this ≠ X (failAc) and myOther ≠ X (failAc) then
        begin
                X (transfer) (1000000, myOther, mine)
                ! Get the wife's money
        end else ...
        ...
end
```

Fig. 22. Using the auto teller

An important difference between the procedure, *transfer*, and the others is that it is defined over more than one object of the witness type. Although the witness type is abstracted over, the values are bound to the same definition, and so are restricted to being the same implementation type. The procedure is written over two values of the same type. If it were not, a type checking error would be detected in the attempt to create the abstract type.

The *transfer* procedure 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, such as this one, are not possible to write without changing the original interface.

5.3 Viewing mechanisms

Viewing mechanisms are traditionally used to provide security and information hiding. Indeed, in some relational database systems, such as INGRES [81], a relational viewing mechanism is the only security mechanism available. A view relation is one which is defined over others to provide a subset of the database, usually at the same level of abstraction. A slightly higher level may be achieved by allowing relations to include derived data, for example, an age field in a view might abstract over a date of birth field in the schema.

Protection provided by view relations is often restricted to simple information hiding by means of projection and selection. For example, if a clerk is not permitted to access a date of birth field, then the projected data may contain all the attributes with the exception of this one. If the clerk may not access any data about people under the age of twenty-one, then the view relation will be that formed by the appropriate selection.

Read-only security may be obtained in some database systems by restricting updates on view relations. Although this restriction is normally due to conceptual and implementation problems, rather than being a deliberate feature of a database system design, it may be used to some effect for this purpose. Some systems, for example IMS [82], go further than this, and the database programmer can allow or disallow insertions, deletions, or modifications of data in view relations. This allows a fine grain of control for data protection purposes.

Views on persistent data may be constructed using abstract data types in Napier88. The technique stores the raw data in the persistent environment and allows access by two mechanisms. The first kind of access is by password where the database administrator may gain access to the raw data on presenting the correct password. The second kind of access is through an abstract data type. The database administrator prepares these abstract data types by placing the raw data and the procedures that operate over it in the interface of an abstract data type. These views may then be stored in the persistent environment for others to use. Data may appear in more than one view and indeed in bulk data types, such as images, overlapping views on the same values are possible.

The first step in constructing views is placing the raw data and operations in an environment. Figure 23 shows how this may be done through the *createAutoTellerEnv* procedure which takes an environment as a parameter and places the interface procedures in that environment. The *lookup* data structure could also be safely placed in this environment. A new procedure *sufficient* determines whether a given account contains sufficient funds to allow a given withdrawal.

```
let createAutoTellerEnv = proc (envir : env)
begin
        type account is structure (Balance, Limit : int ; Pin : string )

        let failAc = account (0, 0,"")
        in envir let failAc = failAc

        in envir
        let getAc = proc (accNumber : int ; passwd : string → account)
        begin
                let new = lookup (accNumber)
                if new (Pin) = passwd then new else failAc
        end

        in envir let withdraw = proc (debit : int ; ac : account → bool)
        begin
                let result = ac (Balance) - debit
                if result > ac (Limit) and debit > 0 then
                begin
                        ac (Balance) := result
                        true
                end else false
        end

        in envir let balance = proc (ac : account → int) ; ac (Balance)

        in envir let olimit = proc (ac : account → int) ; ac (Limit)

        in envir let transfer = proc (amount : int ; from, to : account)
        if amount > 0 and from (Balance) - amount > from (Limit) do
        begin
                from (Balance) := from (Balance) - amount
                to (Balance) := to (Balance) + amount
        end

        in envir let sufficient = proc (debit : int ; ac : account → bool)
        ! Return whether or not debit pounds
        ! may be withdrawn from account ac.
        ac (Balance) - debit > ac (Limit)

end
```

Fig. 23. Placing the auto teller values in an environment

The second step in creating views is to place the raw data environment in the persistent store. This is done by creating a new environment, using the *createAutoTellerEnv* procedure to place the raw data in it, and encapsulating it in a procedure that will only yield the data on the presentation of the correct password. This final procedure is placed in the *User* environment. The technique is demonstrated in Figure 24.

```
let makeProtectedBank = proc (password : string)
begin
        let ps = PS ()

        project ps as X onto
        env : use X with User, Library : env in
            use Library with Environment : env in
            use Environment with environment : proc (→ env) in
            begin
                    let new = environment () ; let fail = environment ()

                    createAutoTellerEnv (new)

                    let this = proc (attempt : string → env)
                    if attempt = password then new else fail

                    in User let protectedBank = this
            end
        default : {} ! This is executed if the projection fails
end
```

Figure 24. Placing the protected data in the persistent store

To construct a view, the database administrator (the person with the correct password), accesses the raw data and places it in an abstract data type. Figure 25 shows how this may be done for the local teller view. It leaves the view in the *User* environment under the name *localTView*.

```
type localTeller is abstype [absac](
            failAc: absac;
            getAc : proc (int, string → absac);
            withdraw    : proc (int, absac → bool);
            balance     : proc (absac → int);
            olimit : proc (absac → int);
            transfer    : proc( int, absac, absac))

type account is structure (Balance, Limit : int ; Pin : string)

let createLocalAutoTeller = proc ()
begin
        let ps = PS ()

        project ps as X onto
        env : use X with User : env in
            use User with protectedBank : proc (string → env) in
            begin
                    let bank = protectedBank ("Correct Password")
                    use bank with failAc :    account;
                                  getAc :     proc (int, string → account);
                                  withdraw :  proc (int, account → bool);
                                  balance :   proc (account → int);
                                  olimit :    proc (account → int);
                                  transfer :  proc( int, account, account) in
                    begin
                          in User let localTView = localTeller [account]
                          (failAc, getAc, withdraw, balance, olimit, transfer)
                    end
            end
        default : {} ! This is executed if the projection fails
end
```

Fig. 25. Constructing the local teller abstract view

Other views may be created by the same mechanism. Figure 26 shows how a remote teller view might be created. Remember that the remote teller cannot transfer money from one account to another and cannot inspect the balance or overdraft limit of the account. However it can find out if there are sufficient fund to make a withdrawal through the *sufficient* procedure in its interface.

```
type remoteTeller is abstype [absac](
          failAc       : absac
          getAc        : proc (int, string → absac)
          withdraw     : proc (int, absac→ bool)
          sufficient   : proc (int, absac→ bool))

type account is structure (Balance, Limit : int ; Pin : string)

let createRemoteAutoTeller = proc ()
begin
      let ps = PS ()

      project ps as X onto
      env : use X with User : env in
          use User with protectedBank : proc (string → env) in
          begin
               let bank = protectedBank ("Correct Password")
               use bank with failAc :       account;
                             getAc :        proc (int, string → account);
                             withdraw :     proc (int, account → bool);
                             sufficient :   proc (int, account → bool) in
               begin
                    in User let remoteTView = remoteTeller [account]
                             (failAc, getAc, withdraw, sufficient)
               end
          end
      default : {} ! This is executed if the projection fails
end
```

Fig. 26. Constructing the remote teller view

Figure 27 illustrates the overall software technique for creating views of data. The raw data and the procedures that operate over the data are stored in the persistent environment and used to construct views as they are required by applications. The view construction is performed by the database administrator who has access to the data through a procedure closure protected by password. Components of the views may be used to construct other views and thus views of views may be constructed to any level of abstraction.

It is interesting to compare this style of encapsulation and information hiding with that of object oriented database systems. In the latter the raw data may only be viewed through one interface and the information is essentially trapped in the object once instantiated. In this technique, the data is placed in an object (view) dynamically when the data modelling requires it. Thus the encapsulation technique is compliant with, and may respond to, the differing needs of different applications and not with some fixed data model defined *a priori*.

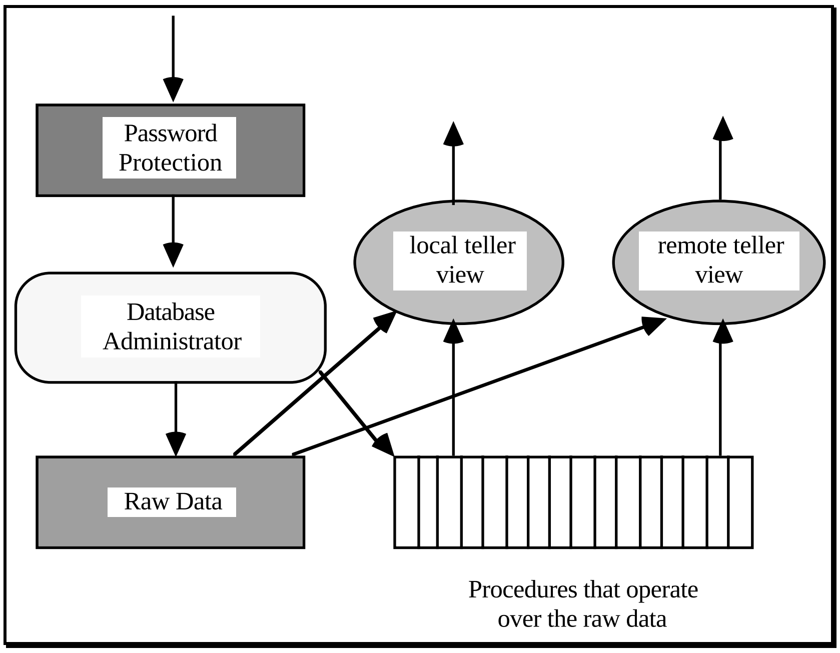


Fig. 27. Views over the persistent store

The final advantage of this style of viewing mechanism is that it is all statically type checked once the views have been constructed.

6 Controlled System Evolution

Evolution is inevitable in persistent systems since the people who use the data, the data itself and the uses to which the data is put, all change with time. Systems which cannot accommodate evolution become obsolete as they can no longer meet the changing needs of the applications and user community that they support. A requirement of a persistent system, like Napier88, which contains data, programs and meta-data, is that the evolution should be controllable from within the system. Most importantly any alteration to the system should not necessarily require total rebuilding. That is, evolution should be incremental since the cost of rebuilding may be prohibitive.

System evolution is caused by changes to the data, the programs which use the data and the meta-data. Changes to program and data with the invariant of fixed meta-data are normally handled by updates to the persistent store. This, however, requires the preparation of new programs, compilation and binding into existing data before updating the persistent store. This non-trivial task is accommodated in Napier88 by a technique called Linguistic Reflection [83] which is described later in this section. A more difficult problem is changes to the meta-data while keeping all the existing programs and data consistent with the semantics of the change [21]. Changes to the schema fall into the following categories:

• additive	extra semantic knowledge is modelled
• subtractive	less semantic knowledge is modelled
• descriptive	the same semantic knowledge is modelled in a different manner

Napier88 provides a number of mechanisms for controlling the evolution of data, programs and meta-data. These include structural type equivalence, the infinite union types **any** and **env**, and linguistic reflection.

6.1 Typing Issues

Separately prepared data and programs require a common mechanism for ensuring that the manner in which they operate is consistent with one another. In a strongly typed system the mechanism is that of type equivalence checking. Two models of type equivalence are in common use: name equivalence and structural equivalence [76]. In name equivalence, the types have the same name defined in a common schema. In structural equivalence, the types have the same structure when compared by an equivalence algorithm. The commonality in this case is the equivalence algorithm. It is shown in [3] that while name equivalence schemes are easier to implement and are more efficient they still have to use structural checks to provide important facilities such as schema merging. On the other hand structural equivalence, generally more flexible and less efficient, can often achieve the same performance as name equivalence. Napier88 uses structural equivalence and therefore requires no common schema.

The infinite union types **any** and **env** may also be used to control system evolution. Remember that the persistent store has a most general dynamically checked type, **any**. To use the values within the store, the dynamic type must be projected onto the specific type for the store. This specific type may in turn contain further instances of **any**. Descriptive changes to the store may be performed by injecting new values into these further instances of **any**. Subsequent use is through the new type descriptions.

This mechanism occurs implicitly in standard database interfaces. When a program opens a database it specifies a schema. During the opening operation the schema used during the program’s compilation is compared with the database’s current schema. Arbitrary changes may have been made to the database using a schema editor between the compilation and this execution. If the schema no longer matches the expectations established at compilation-time then an error is signalled. Thus, internally the run-time system is able to treat the database as having a dynamic type and to perform a dynamic verification that the expected and actual types match.

In general the persistent store is a graph of objects that has one root of type **any**. However, values of type **any** are first class and may also be constituent parts of any other type. Before they can be used with their specific type they also have to be projected onto that type. This allows programs to specify only the part of the schema that they require up to a point of dynamic checking. As a consequence a schema, which is represented by an arbitrarily large collection of mutually referencing types, may scale well since the schema specification is bounded by the dynamic types. Incremental schema changes inject new values into an **any** and the type of the rest of the specification remains unchanged. In addition, where an **any** encapsulates the type, type checking is postponed until required. Hence excessive type checking costs on start-up are avoided. This is illustrated in Figure 28.

```
type Address is structure (name : string ; age : int ; extra : any)

let ps = PS ()

project ps as X onto
Address :
        begin
                let this = X (extra)              ! this is of type any
                type extraInfo is record (idNo : int ; spouse : Address)
                ! Programs not using the extra field do not need to specify
                ! extraInfo
                project this as Y onto
                        ! type check using extraInfo only happens when this is
                        ! executed
                        extraInfo : ...
                ...
                default :     ...
        end
default :       ...
```

Fig. 28. After an incremental schema change

In Figure 28, where the *extra* field is used the specification of the *extraInfo* type is required. Where programs do not use the *extra* field the type *extraFieldInfo* does not have to be declared. Thus only part of the type structure need be specified, the part of interest to the program, as shown in Figure 29.

```
type Address is record (name : string ; age : int ; extra : any)

let ps = PS ()

project ps as X onto
Address :       ...
default :       ...
```

Fig. 29. A partial use of types

The use of dynamic types allows the data model to evolve without recompiling all the programs that refer to the data. For example, if the *extraInfo* type is altered, then only programs that use that type need be altered. Thus, explicit dynamic types allow partial specification of the overall structure of the data (schema) and facilitate the evolution of the data, without having to alter programs that do not make use of the evolutionary changes.

6.2 Type Safe Linguistic Reflection

Type safe linguistic reflection is defined in [83] as the ability of a running program to generate new program fragments and to integrate these into its own execution. This is the basis for system evolution in the Napier88 system.

Napier88 uses run-time linguistic reflection [84-86] which is concerned with the construction and binding of new components with existing components in an environment. The technique involves the use of a compiler that can be called dynamically to compile newly generated program fragments, and a linking mechanism to bind these new program fragments into the running program. Type checking occurs in both compilation and binding. The compiler itself is a value in the persistent store and may be called as a procedure by any program.

Type safe linguistic reflection has been used to attain high levels of genericity [87, 88] and accommodate changes in systems [84, 89]; two examples of these are given below. It has also been used to implement data models [62, 63, 90], optimise implementations [91-93] and validate specifications [94, 95]. The importance of the technique is that it provides a uniform mechanism for software production and evolution. A formal description of linguistic reflection is given in [83].

The example in Figure 30 shows a simple generator which produces code to write out the value of a named string field of a given record. Although somewhat contrived in order to keep the example small, this does demonstrate a problem which requires reflection. The difficulty for non-reflective solutions is that the record field name is not known until run-time, while for the record dereference operation the field name must be known at compile-time to allow static type checking. The reflective approach gets around this by performing compilation once the field name is input, so that the field name is known statically with respect to that compilation.

The program starts by defining representations for code fragments and for type representations within the language. For simplicity here string code representations are used; another possibility would be to use parse trees. Details of the type representations are omitted here.

The generator *writeFieldGen* is then defined. It is a procedure that takes as parameters a record, injected into the infinite union type *any*, and a representation of a field name. The result is the representation of a code fragment. The infinite union type is used for the record parameter so that the generator can accept records of any type.

Inside the generator the first step is to obtain a representation of the type of the record, using the standard procedure *getTypeRep*. The generator then performs a series of checks: that the first parameter is indeed of a record type; that the record type contains the required field name; and finally that field has type *string*. If any of these checks fails an error is reported and an empty code fragment returned, otherwise the result code is formed by concatenating together a number of components. The result represents a procedure which takes a record of the appropriate type as its parameter and writes out the field. The first part of the result code is a local definition of the record type so that it can be used in the procedure header. This involves transforming the type representation into a code fragment, performed by the procedure *typeRepToString*. The rest of the result contains the procedure header and the body which simply dereferences the record and writes out the field.

```
type CodeRep is string ! For simplicity.
type TypeRep is ...

let nilCodeRep = "" ; let newline = "'n"

let ps = PS ()

project ps as X onto
env :   use X with User, Library : env in
        use Library with IO, Reflection : env in
        use IO with writeString : proc( string ) in
        use Reflection with getTypeRep : proc (any → TypeRep);
                    isRecord :          proc (TypeRep → bool);
                    containsField :     proc (TypeRep, CodeRep → bool );
                    fieldType :         proc (TypeRep, string → TypeRep );
                    isString :          proc (TypeRep → bool );
                    typeRepToString :   proc (TypeRep → string ) in

in User
let writeFieldGen = proc(aRecord : any ; fieldName : CodeRep → CodeRep)
begin
    ! Get representation of type.
    let t = getTypeRep (aRecord)

    ! Check that the any contains a record.
    if isRecord (t) then
        ! Check that the record type contains the given field.
        if containsField (t, fieldName) then
            ! Check that the field has type string.
            if isString (fieldType (t, fieldName)) then
                ! The source code produced.
                "type RecordType is " ++ typeRepToString (t) ++ newline ++
                "proc (instance : RecordType)" ++ newline ++
                "writeString (instance (" ++ fieldName ++ "))"
            else{writeString ("field is not of type string"); nilCodeRep}
        else{writeString ("type does not contain given field") ; nilCodeRep}
    else{writeString ("not a record type") ; nilCodeRep}
end
default : {}
```

Fig. 30. A reflective generator

Figure 31 shows an example use of the generator. A particular record type, *Person*, and an instance, *ron*, are defined. The user is then prompted to enter a field name as a string. The generator is then called, passing it the record instance and the field name representation. If no errors occur during generation, the generated code is then compiled with the standard procedure *compile*. Since the type of the result of a compilation is not known in advance, *compile* returns an *any*. This is then projected onto the expected type which is a procedure that takes a single parameter of type *Person*. If the projection matches the compiled procedure is then available for use as *writeAddress*. Otherwise the compilation has either failed or returned a result of a different type: this implies an error in the definition of the generator.

```
type Person is structure (name, address : string ; age : int)
type CodeRep is string

let ps = PS ()

project ps as X onto
env : use X with Library, User : env in
        use Library with Compiler, IO : env in
        use Compiler with compile : proc (CodeRep → any) in
        use IO with readString : proc (→ string);
                    writeString : proc (string) in
        use User with writeFieldGen : proc (any, CodeRep → CodeRep) in
begin
    let ron = Person ("ron”, “8 trinity place”, 42)

    writeString ("which field?")
    let theFieldName = readString ()

    let writeAddressSource = writeFieldGen (any (ron), theFieldName)

    if writeAddress ~= nilCodeRep do
    begin
        project compile (writeAddressSource) as writeAddress onto
            proc (Person)  : writeAddress (ron)
        default            : writeString ("error in generated code")
    end
end
default : {}
```

Fig. 31. Use of a generator

To save space a number of simplifications have been made in this example. The most significant of these is the omission of details of binding to undefined identifiers in the generated code, in this case *writeString*. In reality the generated code fragment would also contain a specification of its location and type in the store.

Binding to existing values in the persistent store is a particular case of the more general problem of specifying a program's execution environment. It might be desirable, for example, for an identifier in a generated code fragment to be bound to a local value created by the generator. This might be achieved by generating code to create a copy of the value, or by storing the value at a location in the persistent store and generating code to bind to that location. Neither is entirely satisfactory. Another approach is to allow generators to produce hyper-programs, so that the generated code representation may contain a direct link to the value [85].

The detection and reporting of errors in both generators and generated code poses many challenges, in particular giving the user intelligible reports about errors which occur in generated code, of whose very existence the user may be unaware.

The example showed the definition and use of a generator within a single program. Alternatively a generator may be defined and made available in the persistent store, from where it is used many times in different environments. In addition, the compiled result obtained from a particular use of the generator may itself be stored and used repeatedly. In this way the costs of generation and compilation are amortised over many uses.

7 Concurrency Control and Transactions

Traditionally the database and programming language communities have taken different approaches to concurrency control. In programming languages, concurrency control is based upon the concept of the co-ordination of a set of co-operating processes by synchronisation. Language constructs such as semaphores [96], monitors [97], mutual exclusion [98], path expressions [99] and message passing [100] have been provided to support this concept. By contrast, in databases, concurrency is viewed as a system efficiency activity which allows parallel execution and parallel access to the data. However, each database process may have to suffer the indignity of abortion in order to sustain the illusion of non-interference. The key concept in databases is that of serialisability [101] which has led to the notion of atomic transactions [101, 102] supported by locking [101] or optimistic concurrency control methods [102].

In both cases the user must attempt to understand the computations in terms of some global cohesion. In programming languages the emphasis is on synchronisation and the overall cohesion is understood in terms of the conflation of all the synchronisations. In database systems, global cohesion is understood in terms of the concept of serialisability [101] but includes failure semantics such as abortion.

Figure 32, taken from [26], illustrates a spectrum of understandability from the points of view of programming language and database users.

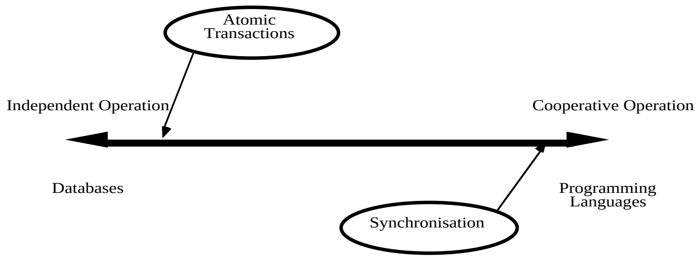


Fig. 32. A spectrum of understandability

Figure 32 illustrates that databases tend to use atomic transactions to enforce isolation rather than co-ordinated sharing. Programming languages promote co-operation. Thus in integrating databases and programming languages, the designer must unify these established and provenly useful positions. The impetus does not altogether come from persistence however, since languages that support atomic transactions and databases that require non-serialisable and designer transactions [103-105] have been identified as necessary by their respective communities.

Concurrency control facilities have not been built into the Napier88 language. Instead they are provided by a number of mechanisms in the persistent environment. This design decision is in line with the V-shape architecture described earlier in which concurrency control is provided at the highest possible level in order to promote flexibility. To facilitate co-operative concurrency a thread abstract data type is provided for concurrent execution and a semaphore package for synchronisation. For competitive and designer transactions, the Napier88 system uses CACS specifications [27, 106] which map onto different store implementations. Thus the implementation technology is tailored to the application. It is beyond the scope of this paper to describe the CACS specification method and therefore we concentrate on the use of threads and semaphores.

The thread package is contained in the *Concurrency* environment and has the following type.

```
type ThreadPack is abstype [thread] (start : proc (proc () → thread) ;
            getCurrentThread : proc (→ thread );
            getAllThreads : proc (→ *thread);
            kill, restart, suspend : proc (thread);
            getStatus : proc (thread → string);
            getParent : proc (thread → variant (present : thread ; absent : nil))
```

The package contains procedures to start a thread, to find the identity of the executing thread, to kill, restart and suspend threads. To start a thread, the start procedure is given as a parameter a second procedure that will execute as the thread. The start procedure returns the identity of the started thread. Control of the thread may be performed through this identity.

The *Concurrency* environment also provides a procedure that takes an integer parameter which is the initial value of the semaphore and returns two procedures, *wait* and *signal*, within a structure that operate over the semaphore.

The use of threads and semaphores is illustrated by a solution to the Dining philosophers problem. There are five philosophers each requiring two forks to eat. The action of obtaining a fork is atomic and therefore protected by a binary semaphore. To avoid deadlock a philosopher must first enter the dining room which is protected by a semaphore with an initial value of four, thereby ensuring that no more than four philosophers may enter at the same time. This is the Butler solution.

The forks are modelled by a vector of semaphore packages where each element of a vector contains a structure containing two synchronisation procedures. The vector is initialised using the *forkSemaphore* procedure. For each element of the vector the procedure is called with the index of the element as a parameter. The result of the procedure is used to initialise the element. Thus the following code segment would initialise each element of the *forks* procedure to a structure containing two procedures implementing a binary semaphore.

```
let forkSemaphore = proc (i : int → Semaphore)
                    semaphoreGen (1)

let forks = vector 0 to 4 using forkSemaphore
```

Figure 33 gives the total solution. The solution uses a number of formatting and I/O procedures from the persistent environment.

```
type ThreadPack is abstype [thread] (start : proc (proc () → thread) ;
            getCurrentThread : proc (→ thread );
            getAllThreads : proc (→ *thread);
            kill, restart, suspend : proc (thread);
            getStatus : proc (thread → string);
            getParent : proc (thread → variant (present : thread ; absent : nil))
type Semaphore is structure (wait, signal : proc ())

project PS() as X onto
env :   use X with Library : env in
        use Library with Concurrency, Format, IO : env in
        use Format with iformat : proc (int → string) in
        use IO with writeString : proc (string) in
        use Concurrency with threadPackage : ThreadPack ;
            semaphoreGen : proc (int → Semaphore) in
begin
    use threadPackage as X [thread] in
    begin
        let room := semaphoreGen (4)
        let forkSemaphore = proc (i : int → Semaphore) ; semaphoreGen (1)
        let forks = vector 0 to 4 using forkSemaphore
        let philosopherGenerator = proc (i : int → thread)
        begin
            let this = "Philosopher " ++ iformat (i)
            let philosopher = proc ()
            while true do
            begin
                writeString (this ++ " is thinking'n") ! Think
                room (wait) () ; writeString (this++ " has entered the room'n")

                forks (i, wait) () ; writeString (this ++ " has one fork'n")
                forks ((i + 1) rem 5, wait) ()
                writeString (this ++ " has two forks and is eating'n")
                forks (i, signal) ()
                writeString (this ++ " has put down one fork'n")
                forks ((i + 1) rem 5, signal) ()
                writeString (this ++ " has put down the second fork'n")

                room (signal) () ; writeString (this ++ " has left the room'n")
            end
            let t = X (start) (philosopher) ; writeString (this ++ " is born'n")
        end
        let philosophers = vector 0 to 4 using philosopherGenerator
    end
end
default : {}
```

Fig. 33. The dining philosophers with annotation

The method of storing threads in the persistent store is the same as for any other data value. That is the thread must be reachable from the root of persistence.

8 Programming within the Persistent Environment

Napier88 is a complete self-contained persistent programming system. As such it supports the use of software throughout its life cycle. For this the system provides an interactive programming environment which is implemented in Napier88 itself, together with facilities for composing, executing and storing persistent programs.

8.1 The Standard Library

In common with many programming systems, Napier88 is supplied with a library of pre-written code and values. Since it is an orthogonally persistent system, this library is supplied as a populated persistent store, as a collection of persistent procedures [41]. The programmer uses the library facilities by writing programs which bind to the appropriate components and manipulate them. Other components in the populated store are used by the system to support its own activities and are not directly accessible by users.

The persistent store may be accessed from Napier88 programs by calling the procedure *PS* as described in Section 3. In the standard release store the persistent root is an environment initially containing the following environments:

Table 2. Standard store contents

name	**environment contents**
Error	error handling procedures which are called when errors occur during the execution of Napier88 programs
External	facilities provided by other sites
Library	standard procedures and other data which may be used in Napier88 programs
User	available for user data

The initial structure of *Error* and *Library* is standardised, whereas the contents of *User* and *External* are specific to a particular installation. The items in the library include procedures for:

- compiling Napier88 programs;
- browsing the persistent store;

- performing I/O and arithmetic;
- constructing graphical user interfaces;
- controlling concurrent threads;
- accessing other Napier88 stores; and
- other utilities.

The library also includes data values which may be updated in order to modify the default behaviour of certain procedures. The initial environment structure of the standard library was shown in Figure 5.

The names of most of the environments in the standard library should be self-explanatory. Some of the more significant are:

Compiler: this environment contains procedures which provide various interfaces to the Napier88 compiler. The simplest is a procedure which takes a string representing a Napier88 program and returns either an error message or a procedure which will execute that program.

Concurrency: this environment contains procedures to manipulate light-weight threads and semaphores [35].

Distribution: this environment contains procedures which can be used to scan the contents of other Napier88 stores and to copy values from them to the local store. A low-level socket based communication protocol between stores is also supported [35].

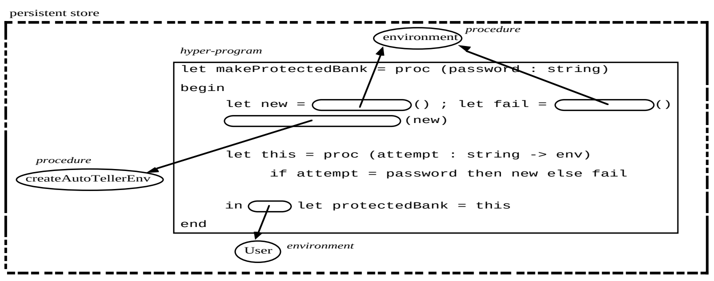
InteractiveEnvironment: this environment contains procedures which provide the interactive programming system described in Section 8.3.

Win: this environment contains procedures for building user interfaces, including window managers, text editors and standard user interface widgets.

8.2 Hyper-Programming

One way for a Napier88 program to use a library component is for the program to contain a textual specification of the component's expected type and location in the persistent store, as described in Section 3. This specification is then checked against the actual state of the store when the program is executed; a run-time error occurs if the two do not match. The activities required of the programmer are thus firstly to discover the type and location of the required library component, and secondly to write down textual descriptions of these in the program.

Since program representations may be held in the persistent store together with the rest of the persistent data, an alternative programming style called *hyper-programming* is possible. With this style the textual descriptions of library components are replaced by direct links to the components themselves embedded within the program representations [20]. The example in Figure 34 shows the code of Figure 24 as a hyper-program.



```
let makeProtectedBank = proc (password : string)
begin
      let new = ( ) ( ) ; let fail = ( ) ( )
      ( ) (new)

      let this = proc (attempt : string -> env)
            if attempt = password then new else fail

      in ( ) let protectedBank = this
end
```

Fig. 34. A hyper-program

The links embedded in the hyper-program are represented by non-textual tokens to allow them to be distinguished from the surrounding text. Note that names for the linked components are no longer necessary. For clarity the components are labelled in the diagram, but these names are not part of the semantics of the hyper-program.

Hyper-programming facilities are used in the persistent programming environment to reduce the need for the programmer to supply textual descriptions of library components. Instead the programmer identifies a component, by means to be described, and then links it into the program under construction. This can lead to a significant reduction in the amount of code written. A hyper-program editor which displays links as light-buttons embedded in the text is supplied.

The hyper-program notation also provides a convenient user-interface representation for procedures which contain free variables: each free variable is denoted by a light-button in the same way as a linked library component. This enables the system to display the source code of any procedure, even if it has encapsulated state. A flag in the Napier88 compiler specifies whether to retain the source code for a procedure being compiled: if so, the hyper-program source representation is linked to the procedure object, from which it may be later retrieved and displayed. By default this flag is on, so all the library components have their source code attached automatically.

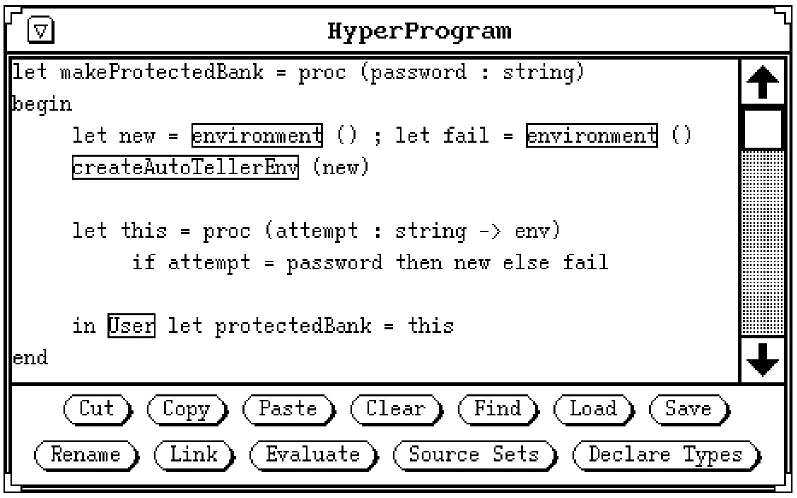
8.3 The Napier88 Programming Environment

The programming environment provides several varieties of window:

- hyper-program editor windows;
- a compilation error display window;
- a browser window; and
- declaration set windows.

8.3.1 Editing Hyper-Programs

Hyper-program windows may be created by selecting *New Editor* from the background menu. Each window contains a hyper-program text editing area, a scroll bar and a row of light-buttons. An example of a hyper-program window is shown in Figure 35:



```
let makeProtectedBank = proc (password : string)
begin
     let new = environment () ; let fail = environment ()
     createAutoTellerEnv (new)

     let this = proc (attempt : string -> env)
          if attempt = password then new else fail

     in User let protectedBank = this
end
```

Fig. 35. A hyper-program window

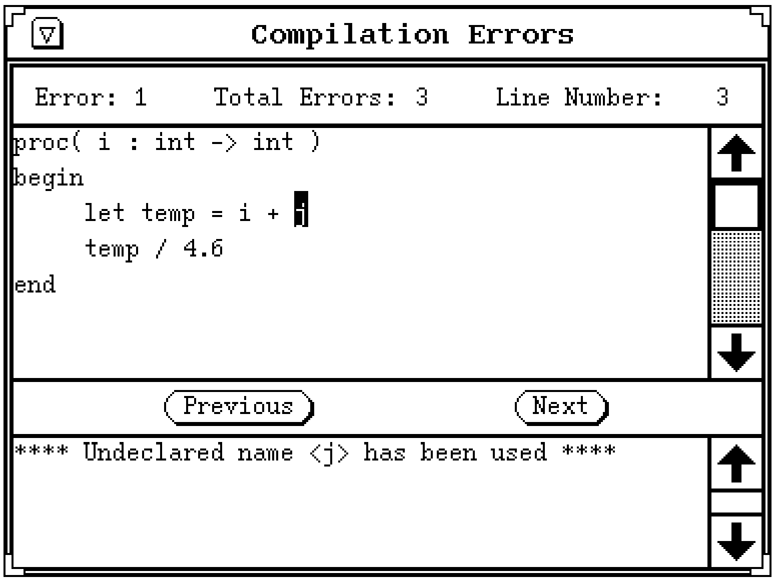
The editor operations include the usual editing functions together with those described in Table 3:

Table 3. Light-button operations in hyper-program window

operation	**action**
Link	This inserts a hyper-program link to the currently selected value, location or type. A light-button representing it is inserted into the hyper-program text. The label is the name, if any, associated with the selection. The value, location or type associated with a button can be displayed in the browser window by clicking on the button.
Evaluate	This attempts to compile the currently selected hyper-program text, executes the result if successful, and displays any result in the browser window. If a compilation error occurs the compilation error window is displayed.
Source Sets	This displays a dialogue allowing the source declaration sets to be set .
Declare Types	This attempts to compile the currently selected hyper-program text and adds any type declarations in scope at the end of the compilation to a selected declaration set.

Components to be linked into a hyper-program are identified by traversing links in the browser window, to be described more fully in Section 8.3.2. This relies on the programmer having some prior knowledge of the library structure. Further development of tools to assist the programmer in finding components is needed [107, 108].

The compilation error window is displayed when compilation errors are encountered in a hyper-program. One sub-window shows the source code with the region of the first error highlighted. The second sub-window shows a message describing the error. When multiple errors are detected the *Next* and *Previous* buttons can be used to scroll through the errors. An example is shown in Figure 36:



```
Error: 1     Total Errors: 3     Line Number:    3
```

```
proc( i : int -> int )
begin
     let temp = i + j
     temp / 4.6
end
```

```
**** Undeclared name <j> has been used ****
```

Fig. 36. The compilation error window

8.3.2 The Browser

The browser window is used to display representations of values produced by the evaluation of hyper-programs. The root of the persistent store can be displayed by selecting *Show PS* from the background menu. The form in which a value is represented depends on the type of the value. Integers, reals, strings and booleans are written to the output window.

Each window displayed in the browser window can be selected or deselected by clicking on the border. If the window is not already selected it becomes selected and any other selected windows are deselected. When a window is selected the corresponding value is also considered to be selected: this is of relevance when inserting links into hyper-programs.

To show an environment the browser displays a menu window containing an entry for each binding in the environment. For base type values the corresponding entry shows the type while for instances of constructed types only the type constructor is shown. An example is shown in Figure 37:

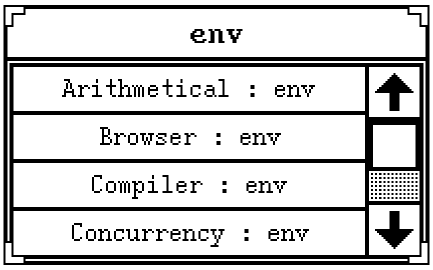


Fig. 37. An environment menu

An environment menu entry may be either selected or displayed, depending on the mouse button used to click on it. Displaying an entry results in the value of the corresponding environment binding being displayed in the browser. If the value is of such a type that a new window is displayed for it, an arrow is drawn from the menu entry to the new window as shown in Figure 38.

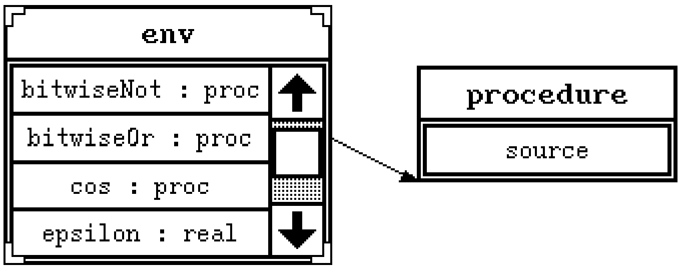


Fig. 38. Link from environment location to value

Structures are displayed in the same way as environments, with a menu entry for each field, as illustrated in Figure 39:

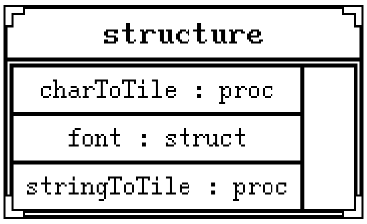
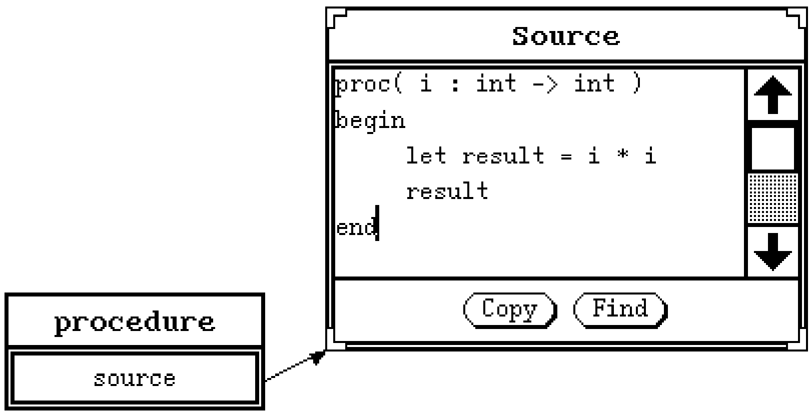


Fig. 39. A structure menu

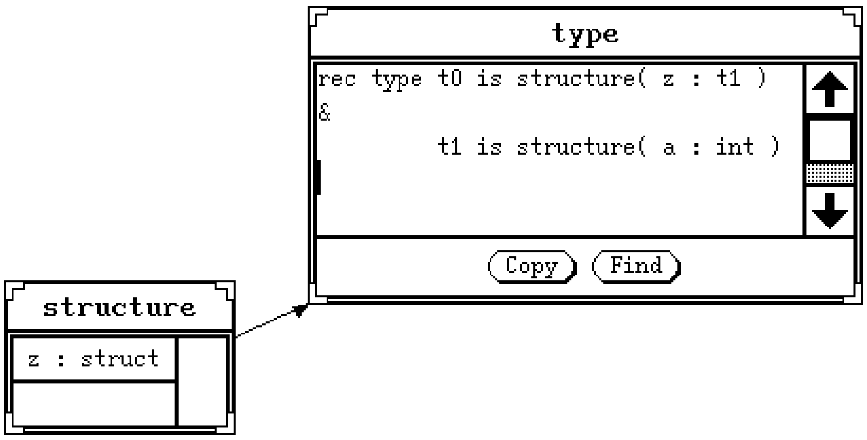
To show a procedure the browser displays a menu with a single entry *source*. When this entry is clicked on the browser displays a hyper-program window containing the source code for the procedure. The source code may be copied but not altered. An example is shown in Figure 40:



```
proc( i : int -> int )
begin
     let result = i * i
     result
end
```

Fig. 40. A procedure window

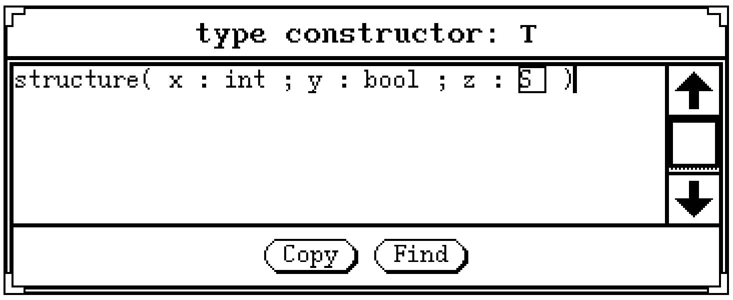
A representation of the type of a value in the browser window may be obtained by selecting the corresponding window and selecting *Show Type* from the background menu. The browser displays a window containing a canonical string representation of the value's type. An example is shown in Figure 41:



```
rec type t0 is structure( z : t1 )
&
        t1 is structure( a : int )
```

Fig. 41. A type representation

The browser also displays a representation of a type linked into a hyper-program when the corresponding light-button in the hyper-program window is pressed. In this case the representation may be a canonical string as above or, where type constructor information is available, the original source code is displayed as a hyper-program fragment. An example of a type constructor source representation, with a hyper-program link to a component type *S*, is shown in Figure 42:



```
structure( x : int ; y : bool ; z : S )
```

Fig. 42. A type constructor representation

8.3.3 Declaration Sets

For convenience the user may create *declaration sets* containing named values, locations and types to use in future program evaluation. Each declaration set has a unique name and may be thought of as forming an additional outer scope for a program. Free identifiers in a program are resolved by scanning the declaration sets associated with the program.

A type entry in a declaration set may represent either a type only, or a type constructor. Which is obtained depends on the method used to create the entry. Both type and type constructor names may be used as type denotations in programs, but only type constructor names may be used to construct instances of types.

The declaration sets model is based on a number of earlier systems: Napier88 Release 1.0 [109]; ABERDEEN [110]; and a previous version of the Napier88 programming environment [111].

The operations on declaration sets are:

- create a new declaration set;
- delete a declaration set;
- add a value, location or type to a declaration set;
- display the contents of a declaration set; and
- choose an ordered list of declaration sets to use for compilation.

The contents of a particular declaration set may be displayed by pressing the *Show* button in the main declaration sets menu. An example is shown in Figure 43:

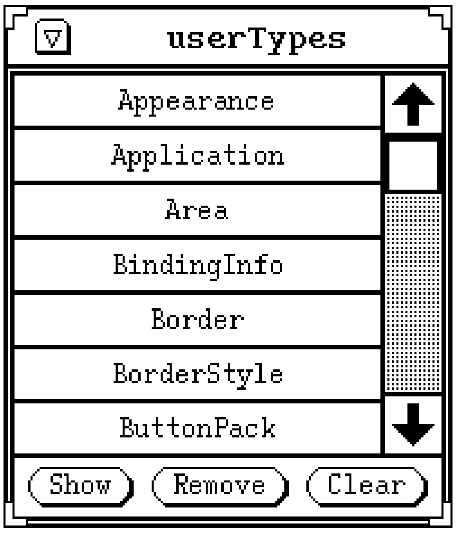


Fig. 43. A declaration set menu

Each menu contains a list of the entries in that declaration set. An entry may be displayed by clicking on *Show* or linked into a hyper-program by selected it and clicking on *Link* in the editor.

The user may associate a particular combination of declaration sets with a hyper-program editor. These declaration sets are then used in evaluating program fragments in that editor. Declaration sets may be added to an editor's list by clicking on *Source Sets*. This displays a dialogue as shown in Figure 44:

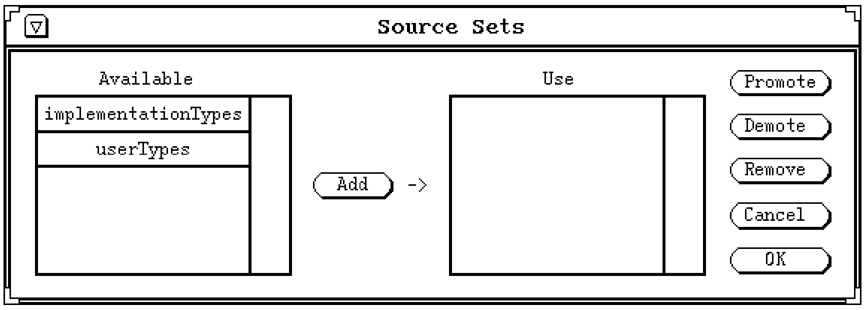


Fig. 44. Dialogue for setting source declaration sets

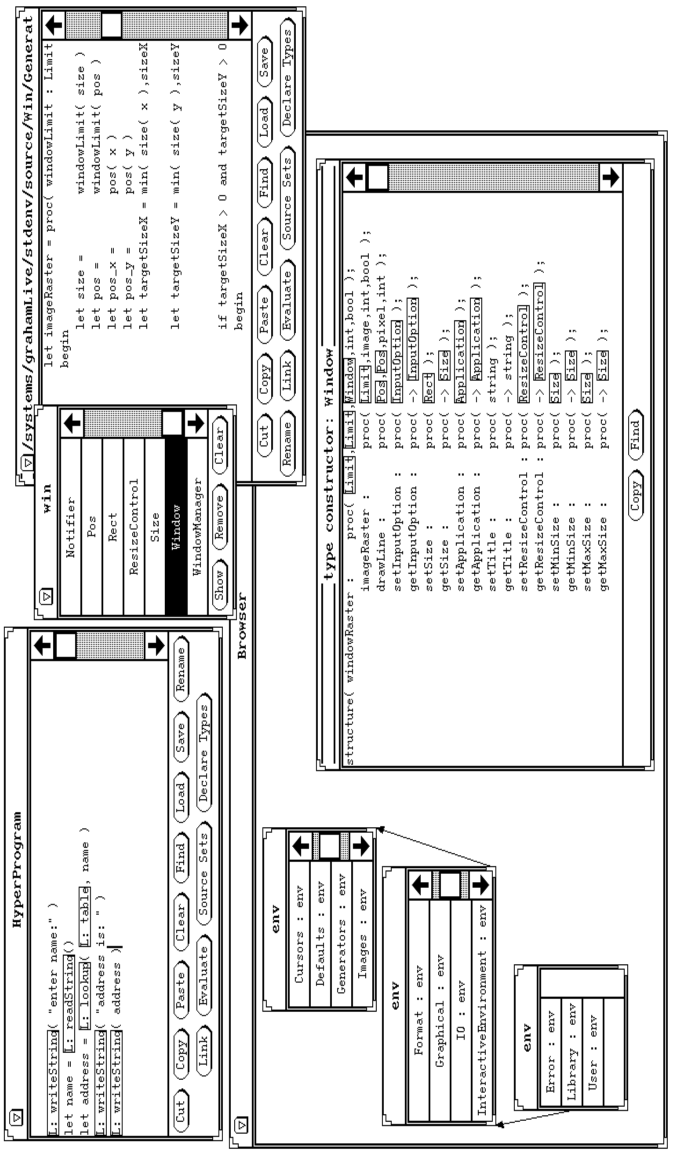
The *Available* list on the left shows all the existing declaration sets. The *Use* list on the right shows those currently associated with the editor, scope level increasing down the list. If two declaration sets associated with an editor both contain an entry with the same name, the one in the declaration set nearer the top of the list will mask the other. This is analogous to normal scoping rules.

8.3.4 Multiple Users

More than one programming environment session may be active simultaneously. The name of the initial sessions can be specified as parameters to the system startup command. Thus multiple users may operate in a given persistent store simultaneously.

Programming environment windows persist between sessions of the programming environment. When a session is shut down the positions and sizes of the windows are recorded and restored when it is next started up. Each session contains its own browser, output window, compilation error window, hyper-program windows etc. No particular concurrency control scheme is enforced: for flexibility this is left to applications as described in Section 7. Thus by default an update to the persistent store by one user is immediately visible to others.

Figure 45 shows an example programming environment session. The two windows at the top-left and top-right (when viewed side-on) are hyper-program editors. Currently the editor at the top-right happens to contain only text. The menu between the two hyper-program editors shows the contents of the declaration set *win*: a set of types used for user interface programming. One of the types, *Window*, has been selected and displayed in the browser window in the lower half of the screen. Since this type has source information attached the browser is able to display the original source code from which the type derives. The source contains hyper-program links to other types used in the definition. The user could click on one of the links to display that component type. The menus on the left of the browser window show a series of environments accessible from the persistent root.


```
HyperProgram
L: writeString( "enter name:" )
let name = L: readString()
let address = L: lookup( L: table, name )
L: writeString( "address is: " )
L: writeString( address )
```

```
/systems/grahamLive/stdenv/source/Win/Generat
let imageRaster = proc( windowLimit : Limit
begin
        let size =          windowLimit( size )
        let pos =           windowLimit( pos )
        let pos_x =         pos( x )
        let pos_y =         pos( y )
        let targetSizeX = min( size( x ),sizeX

        let targetSizeY = min( size( y ),sizeY


        if targetSizeX > 0 and targetSizeY > 0
        begin
```

```
type constructor: Window
structure( windowRaster :      proc( Limit,Limit,Window,int,bool );
           imageRaster :       proc( Limit,image,int,bool );
           drawLine :          proc( Pos,Pos,pixel,int );
           setInputOption :    proc( InputOption );
           getInputOption :    proc( -> InputOption );
           setSize :           proc( Rect );
           getSize :           proc( -> Size );
           setApplication :    proc( Application );
           getApplication :    proc( -> Application );
           setTitle :          proc( string );
           getTitle :          proc( -> string );
           setResizeControl :  proc( ResizeControl );
           getResizeControl :  proc( -> ResizeControl );
           setMinSize :        proc( Size );
           getMinSize :        proc( -> Size );
           setMaxSize :        proc( Size );
           getMaxSize :        proc( -> Size );
```

Fig. 45. A programming environment session

8.4 Implementation Issues

The programming environment is entirely implemented in Napier88, except for a very few components (such as parts of the compiler) which need to perform operations below the type system level. The implementation relies heavily on the provision of orthogonal persistence to store the procedure components which make up the system. Much use is also made of the graphics facilities provided by Napier88, which allow the manipulation of graphical data as first class values [46].

Since not all the components present in the standard populated store are designed to be accessible to users, protection mechanisms are necessary. Both 1st-order and 2nd-order information hiding are used. Some components are hidden in the closures of the procedures which use them, and have no direct access path from the root of persistence. This prevents users from linking to them (although it may be necessary to restrict access to the hyper-program source code of the procedures in which they are encapsulated). 2nd-order information hiding is used to allow users restricted access to components. For example the user may obtain a reference to a representation of the type of a given value in the browser, but is prevented from discovering any information about its internal structure since the representation is a witness of an abstract data type. All the user can do with the representation is pass it to some library procedure which operates on type representations. Password protection is used to restrict access to the raw type representations which can only be accessed by system components.

9 Acknowledgements

This work was supported by the Alvey funded SERC grant GR/D 43266, a grant from International Computers Ltd (ICL), SERC grant GR/F 02953, ESPRIT II Basic Research Action 3070 — Fide$_1$ and ESPRIT III Basic Research Action 6309 — FIDE$_2$. Richard Connor is supported by EPSRC Advanced Fellowship B/94/AF/1921.

10 References

1. Atkinson MP, Morrison R, Pratten GD. Designing a Persistent Information Space Architecture. In: Proc. 10th IFIP World Congress, Dublin, 1986, pp 115-120

2. Connor RCH. The Napier Type-Checking Module. Universities of Glasgow and St Andrews Report PPRR-58-88, 1988

3. Connor RCH, Brown AB, Cutts QI, Dearle A, Morrison R, Rosenberg J. Type Equivalence Checking in Persistent Object Systems. In: Dearle A, Shaw GM, Zdonik SB (ed) Implementing Persistent Object Bases, Principles and Practice, Proc. 4th International Workshop on Persistent Object Systems, Martha’s Vineyard, USA. Morgan Kaufmann, 1990, pp 151-164

4. Connor RCH. Types and Polymorphism in Persistent Programming Systems. Ph.D. thesis, University of St Andrews, 1990

5. Connor RCH, McNally DJ, Morrison R. Subtyping and Assignment in Database Programming Languages. In: Kanelakis P, Schmidt JW (ed) Database Programming Languages: Bulk Types and Persistent Data, Proc. 3rd International Workshop on Database Programming Languages, Nafplion, Greece. Morgan Kaufmann, 1991, pp 363-382

6. Connor RCH, Morrison R. Subtyping Without Tears. In: Proc. 15th Australian Computer Science Conference, Hobart, Tasmania, 1992, pp 209-225

7. Morrison R, Brown AL, Carrick R, Connor RCH, Dearle A, Atkinson MP. The Napier Type System. In: Rosenberg J, Koch DM (ed) Persistent Object Systems, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia. Springer-Verlag, 1990, pp 3-18

8. Atkinson MP, Lécluse C, Philbrow P, Richard P. Design Issues in a Map Language. In: Kanellakis P, Schmidt JW (ed) Bulk Types & Persistent Data. Morgan Kaufmann, 1991, pp 20-32

9. Connor RCH, Atkinson MP, Berman S, Cutts QI, Kirby GNC, Morrison R. The Joy of Sets. In: Beeri C, Ohori A, Shasha DE (ed) Database Programming Languages, Proc. 4th International Conference on Database Programming Languages (DBPL4), New York City. Springer-Verlag, 1993, pp 417-433

10. Connor RCH, Dearle A, Morrison R, Brown AL. Existentially Quantified Types as a Database Viewing Mechanism. In: Bancilhon F, Thanos C, Tsichritzis D (ed) Lecture Notes in Computer Science 416, Proc. 2nd International Conference on Extending Database Technology, Venice, Italy. Springer-Verlag, 1990, pp 301-315

11. Morrison R, Brown AL, Connor RCH et al. Protection in Persistent Object Systems. In: Rosenberg J, Keedy JL (ed) Security and Persistence, Proc. International Workshop on Security and Persistence, Bremen. Springer-Verlag, 1990, pp 48-66

12. Morrison R, Dearle A, Connor RCH, Brown AL. An Ad-Hoc Approach to the Implementation of Polymorphism. ACM Transactions on Programming Languages and Systems 1991; 13,3:342-371

13. Connor RCH, Dearle A, Morrison R, Brown AL. An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance. In: Proc. OOPSLA'89, New Orleans, Louisiana, 1989

14. Atkinson MP, Buneman OP, Morrison R. Binding and Type Checking in Database Programming Languages. Computer Journal 1988; 31,2:99-109

15. Atkinson MP, Morrison R. Types, Bindings and Parameters in a Persistent Environment. In: Atkinson MP, Buneman OP, Morrison R (ed) Data Types

and Persistence, Proc. 1st International Workshop on Persistent Object Systems, Appin, Scotland. Springer-Verlag, 1988, pp 3-20

16. Morrison R, Brown AL, Dearle A, Atkinson MP. On the Classification of Binding Mechanisms. Information Processing Letters 1990; 34:51-55

17. Morrison R, Connor RCH, Cutts QI, Dunstan VS, Kirby GNC. Exploiting Persistent Linkage in Software Engineering Environments. Computer Journal 1995; 38,1:1-16

18. Morrison R, Brown AL, Carrick R, Connor RCH, Dearle A, Atkinson MP. Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment. Software Engineering Journal 1987; ,December:199-204

19. Morrison R, Connor RCH, Cutts QI, Kirby GNC. Persistent Possibilities for Software Environments. In: The Intersection between Databases and Software Engineering, Proc. ICSE-16 Workshop on the Intersection between Databases and Software Engineering, Sorrento, Italy. IEEE Computer Society Press, 1994, pp 78-87

20. Kirby GNC, Connor RCH, Cutts QI, Dearle A, Farkas AM, Morrison R. Persistent Hyper-Programs. In: Albano A, Morrison R (ed) Persistent Object Systems, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy. Springer-Verlag, 1992, pp 86-106

21. Connor RCH, Cutts QI, Kirby GNC, Morrison R. Using Persistence Technology to Control Schema Evolution. In: Proc. 9th ACM Symposium on Applied Computing, Phoenix, Arizona, 1994, pp 441-446

22. Atkinson MP, Sjøberg DIK, Morrison R. Managing Change in Persistent Object Systems. In: Proc. JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan, 1993, pp 315-338

23. Morrison R, Connor RCH, Cutts QI, Kirby GNC, Stemple D. Mechanisms for Controlling Evolution in Persistent Object Systems. Journal of Microprocessors and Microprogramming 1993; 17,3:173-181

24. Morrison R, Brown AL, Carrick R, Connor RCH, Dearle A. On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments. In: Dittrich KR (ed) Lecture Notes in Computer Science 334, Proc. 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, Germany. Springer-Verlag, 1988, pp 334-339

25. Morrison R, Barter CJ, Brown AL et al. Language Design Issues in Supporting Process-Oriented Computation in Persistent Environments. In: Proc. 22nd International Conference on System Sciences, Hawaii, 1989, pp 736-744

26. Munro DS, Connor RCH, Morrison R, Scheuerl S, Stemple D. Concurrent Shadow Paging in the Flask Architecture. In: Atkinson MP, Maier D, Benzaken V (ed) Persistent Object Systems, Proc. 6th International Workshop on Persistent Object Systems, Tarascon, France. Springer-Verlag, 1994, pp 16-42

27. Stemple D, Morrison R. Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach. In: Proc. 15th Australian Computer Science Conference, Hobart, Tasmania, 1992, pp 873-891

28. Brown AL, Rosenberg J. Persistent Object Stores: An Implementation Technique. In: Dearle A, Shaw GM, Zdonik SB (ed) Implementing Persistent Object Bases, Principles and Practice, Proc. 4th International Workshop on Persistent Object Systems, Martha’s Vineyard, USA. Morgan Kaufmann, 1990, pp 199-212

29. Brown AL, Cockshott WP. The CPOMS Persistent Object Management System. Universities of Glasgow and St Andrews Report PPRR-13-85, 1985

30. Brown AL, Morrison R. A Generic Persistent Object Store. Software Engineering Journal 1992; 7,2:161-168

31. Brown AL, Mainetto G, Matthes F, Müller R, McNally DJ. An Open System Architecture for a Persistent Object Store. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 766-776

32. Vaughan F, Schunke T, Koch B, Dearle A, Marlin C, Barter C. A Persistent Distributed Architecture Supported by the Mach Operating System. In: Proc. Proceedings of the 1st USENIX Conference on the Mach Operating System, 1990, pp 123-140

33. Koch B, Schunke T, Dearle A et al. Cache Coherence and Storage Management in a Persistent Object System. In: Dearle A, Shaw G, Zdonik SB (ed) Implementing Persistent Object Bases. Morgan Kaufmann, 1990, pp 103-113

34. Brown AL. Persistent Object Stores. Ph.D. thesis, University of St Andrews, 1989

35. Munro DS. On the Integration of Concurrency, Distribution and Persistence. Ph.D. thesis, University of St Andrews, 1993

36. Sjøberg DIK. Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems. Ph.D. thesis, University of Glasgow, 1993

37. Sjøberg DIK, Atkinson MP, Lopes JC, Trinder PW. Building an Integrated Persistent Application. In: Beeri C, Ohori A, Shasha DE (ed) Database Programming Languages, Proc. 4th International Conference on Database

Programming Languages, New York City. Springer-Verlag, 1993, pp 359-375

38. Sjøberg DIK, Cutts QI, Welland R, Atkinson MP. Analysing Persistent Language Applications. In: Atkinson MP, Maier D, Benzaken V (ed) Persistent Object Systems, Proc. 6th International Workshop on Persistent Object Systems, Tarascon, France. Springer-Verlag, 1994, pp 235-255

39. Connor RCH, Cutts QI, Kirby GNC, Moore VS, Morrison R. Unifying Interaction with Persistent Data and Program. In: Sawyer P (ed) Interfaces to Database Systems, Proc. 2nd International Workshop on User Interfaces to Databases, Ambleside, Cumbria, 1994. Springer-Verlag, 1994, pp 197-212

40. Morrison R, Brown AL, Connor RCH et al. The Napier88 Reference Manual (Release 2.0). University of St Andrews Report CS/94/8, 1994

41. Kirby GNC, Brown AL, Connor RCH et al. The Napier88 Standard Library Reference Manual (Release 2.0). University of St Andrews Report CS/94/7, 1994

42. Goldberg A, Robson D. Smalltalk-80: The Language and its Implementation. Addison Wesley, Reading, Massachusetts, 1983

43. PS-algol Reference Manual, 4th edition. Universities of Glasgow and St Andrews Report PPRR-12-88, 1988

44. Morrison R. S-algol Language Reference Manual. University of St Andrews Report CS/79/1, 1979

45. Atkinson MP, Morrison R. Orthogonally Persistent Object Systems. VLDB Journal 1995; 4,3:319-401

46. Morrison R, Brown AL, Dearle A, Atkinson MP. An Integrated Graphics Programming Environment. Computer Graphics Forum 1986; 5,2:147-157

47. Morrison R, Brown AL, Bailey PJ, Davie AJT, Dearle A. A Persistent Graphics Facility for the ICL PERQ Computer. Software—Practice and Experience 1986; 16,4:351-367

48. Albano A, Cardelli L, Orsini R. Galileo: a Strongly Typed, Interactive Conceptual Language. ACM Transactions on Database Systems 1985; 10,2:230-260

49. Matthes F, Müller R, Schmidt JW. Object Stores as Servers in Persistent Programming Environments—The P-Quest Experience. ESPRIT BRA Project 3070 FIDE Report FIDE/92/48, 1992

50. Davie AJT, McNally DJ. Statically Typed Applicative Persistent Language Environment (STAPLE) Reference Manual. University of St Andrews Report CS/90/14, 1990

51. Kirby GNC, Connor RCH, Cutts QI, Morrison R, Munro DS, Scheuerl S. Using the Flask Architecture to Build Distributed Applications. ESPRIT BRA Project 6309 FIDE$_2$ Report FIDE/95/127, 1995

52. Garcia-Molina H. Using Semantic Knowledge for Transaction Processing in a Distributed Database. ACM Transactions on Database Systems 1983; 8,2:186-213

53. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott WP, Morrison R. An Approach to Persistent Programming. Computer Journal 1983; 26,4:360-365

54. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott WP, Morrison R. Progress with Persistent Programming. In: Stocker PM, Atkinson MP, Gray PM (ed) Database, Role and Structure. Cambridge University Press, 1984, pp 245-310

55. Atkinson MP, Buneman OP. Types and Persistence in Database Programming Languages. ACM Computing Surveys 1987; 19,2:105-190

56. Atkinson MP, Chisholm KJ, Cockshott WP. PS-algol: An Algol with a Persistent Heap. ACM SIGPLAN Notices 1982; 17,7:24-31

57. Atkinson MP, Morrison R. Procedures as Persistent Data Objects. ACM Transactions on Programming Languages and Systems 1985; 7,4:539-559

58. Atkinson MP, Morrison R, Pratten GD. A Persistent Information Space Architecture. In: Proc. 9th Australian Computing Science Conference, Australia, 1986

59. Dearle A. Constructing Compilers in a Persistent Environment. In: Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland, 1987

60. Dearle A. On the Construction of Persistent Programming Environments. Ph.D. thesis, University of St Andrews, 1988

61. Wai F. Distribution and Persistence. In: Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland, 1987, pp 207-225

62. Cooper RL. Configurable Data Modelling Systems. In: Proc. 9th International Conference on the Entity Relationship Approach, Lausanne, Switzerland, 1990, pp 35-52

63. Cooper RL. On The Utilisation of Persistent Programming Environments. Ph.D. thesis, University of Glasgow, 1990

64. McCarthy J, Abrahams PW, Edwards DJ, Hart TP, Levin MI. The Lisp Programmers’ Manual. M.I.T. Press, Cambridge, Massachusetts, 1962

65. van Wijngaarden A, Mailloux BJ, Peck JEL, Koster CHA. Report on the Algorithmic Language ALGOL 68. Numerische Mathematik 1969; 14:79-218

66. Strachey C. Fundamental Concepts in Programming Languages. Oxford University Press, Oxford, 1967

67. Tennent RD. Language Design Methods Based on Semantic Principles. Acta Informatica 1977; 8:97-112

68. Cardelli L, Wegner P. On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys 1985; 17,4:471-523

69. Dearle A. Environments: A flexible binding mechanism to support system evolution. In: Proc. 22nd International Conference on Systems Sciences, Hawaii, 1989, pp 46-55

70. Atkinson MP, Morrison R. Integrated Persistent Programming Systems. In: Proc. 19th International Conference on Systems Sciences, Hawaii, 1986, pp 842-854

71. Milner R. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 1978; 17,3:348-375

72. Milner R, Tofte M, Harper R. The Definition of Standard ML. MIT Press, Cambridge, Massachusetts, 1989

73. Demers A, Donahue J. Revised Report on Russell. Cornell University Report TR79-389, 1979

74. Matthews DCJ. Poly Manual. University of Cambridge, 1985

75. Mitchell JC, Plotkin GD. Abstract Types have Existential Type. ACM Transactions on Programming Languages and Systems 1988; 10,3:470-502

76. Albano A, Dearle A, Ghelli G et al. A Framework for Comparing Type Systems for Database Programming Languages. In: Hull R, Morrison R, Stemple D (ed) Database Programming Languages. Morgan Kaufmann, 1989, pp 170-178

77. Dahl O, Nygaard K. Simula, an Algol-Based Simulation Language. Communications of the ACM 1966; 9,9:671-678

78. Hammer M, McLeod D. Database Description with SDM: A Semantic Database Model. ACM Transactions on Database Systems 1981; 6,3:351-386

79. Lorie RA. Physical Integrity in a Large Segmented Database. ACM Transactions on Database Systems 1977; 2,1:91-104

80. Rosenberg J, Henskens F, Brown AL, Morrison R, Munro D. Stability in a Persistent Store Based on a Large Virtual Memory. In: Rosenberg J, Keedy

JL (ed) Security and Persistence, Proc. International Workshop on Security and Persistence, Bremen, 1990. Springer-Verlag, 1990, pp 229-245

81. Stonebraker M, Wong E, Kreps P, Held G. The Design and Implementation of INGRES. ACM Transactions on Database Systems 1976; 1,3:189-222

82. Davies CT. Data Processing Spheres of Control. IBM Systems Journal 1978; 17,2:179-198

83. Stemple D, Stanton RB, Sheard T et al. Type-Safe Linguistic Reflection: A Generator Technology. ESPRIT BRA Project 3070 FIDE Report FIDE/92/49, 1992

84. Dearle A, Brown AL. Safe Browsing in a Strongly Typed Persistent Environment. Computer Journal 1988; 31,6:540-544

85. Kirby GNC, Connor RCH, Morrison R. START: A Linguistic Reflection Tool Using Hyper-Program Technology. In: Atkinson MP, Maier D, Benzaken V (ed) Persistent Object Systems, Proc. 6th International Workshop on Persistent Object Systems (POS6), Tarascon, France. Springer-Verlag, 1994, pp 355-373

86. Kirby GNC. Persistent Programming with Strongly Typed Linguistic Reflection. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 820-831

87. Stemple D, Fegaras L, Sheard T, Socorro A. Exceeding the Limits of Polymorphism in Database Programming Languages. In: Bancilhon F, Thanos C, Tsichritzis D (ed) Lecture Notes in Computer Science 416. Springer-Verlag, 1990, pp 269-285

88. Sheard T. Automatic Generation and Use of Abstract Structure Operators. ACM Transactions on Programming Languages and Systems 1991; 19,4:531-557

89. Dearle A, Cutts QI, Kirby GNC. Browsing, Grazing and Nibbling Persistent Data Structures. In: Rosenberg J, Koch DM (ed) Persistent Object Systems, Proc. 3rd International Workshop on Persistent Object Systems (POS3), Newcastle, Australia. Springer-Verlag, 1990, pp 56-69

90. Cooper RL, Qin Z. A Graphical Data Modelling Program With Constraint Specification and Management. In: Proc. 10th British National Conference on Databases, Aberdeen, 1992, pp 192-208

91. Cooper RL, Atkinson MP, Dearle A, Abderrahmane D. Constructing Database Systems in a Persistent Environment. In: Proc. 13th International Conference on Very Large Data Bases, 1987, pp 117-125

92. Fegaras L, Stemple D. Using Type Transformation in Database System Implementation. In: Kanelakis P, Schmidt JW (ed) 3rd International

Conference on Database Programming Languages. Morgan Kaufmann, 1991, pp 337-353

93. Cutts QI, Connor RCH, Kirby GNC, Morrison R. An Execution Driven Approach to Code Optimisation. In: Proc. 17th Australasian Computer Science Conference (ACSC'94), Christchurch, New Zealand, 1994, pp 83-92

94. Stemple D, Sheard T, Fegaras L. Linguistic Reflection: A Bridge from Programming to Database Languages. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 844-855

95. Fegaras L, Sheard T, Stemple D. Uniform Traversal Combinators: Definition, Use and Properties. In: Proc. 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York, 1992

96. Dijkstra EW. The Structure of the T.H.E. Multiprogramming System. Communications of the ACM 1968; 11,5:341-346

97. Hoare CAR. Monitors: An Operating System Structuring Concept. Communications of the ACM 1974; 17,10:549-557

98. Dijkstra EW. Cooperating Sequential Processes. In: Genuys F (ed) Programming Languages. Academic Press, 1968, pp 43-112

99. Campbell RH, Haberman AN. The Specification of Process Synchronisation by Path Expressions. In: Lecture Notes in Computer Science 16. Springer-Verlag, 1974

100. Brookes SD, Hoare C, Roscoe A. A Theory of Communicating Sequential Processes. Carnegie-Mellon University Report CMU-CS-83-153, 1980

101. Eswaran KP, Gray JN, Lorie RA, Traiger IL. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM 1976; 19,11:624-633

102. Kung HT, Robinson JT. On Optimistic Methods for Concurrency Control. ACM Transactions on Database Systems 1982; 6,2:213-226

103. Nodine MH, Zdonik SB. Co-operative Transaction Hierarchies: Transaction Support for Design Applications. VLDB Journal 1992; 1,1:41-80

104. Sutton S. A Flexible Consistency Model for Persistent Data in Software-Process Programming. In: Dearle A, Shaw GM, Zdonik SB (ed) Implementing Persistent Object Bases, Principles and Practice, Proc. 4th International Workshop on Persistent Object Systems, Martha’s Vineyard, USA. Morgan Kaufmann, 1990, pp 305-319

105. Ellis CA, Gibbs SJ. Concurrency Control in Groupware Systems. In: Proc. ACM-SIGMOD International Conference on Management of Data, Portland, Oregon, 1989, pp 399-407

106. Morrison R, Barter CJ, Connor RCH et al. Concurrency Control in Process Models. IOPENER 1993; 2,1:11-12

107. Brown JC. A Library Explorer for the Napier88 Glasgow Libraries. M.Sc. thesis, University of Glasgow, 1993

108. Glasgow Workshop. The Glasgow Persistent Workshop: User Documentation. ESPRIT BRA Project 6309 $FIDE_2$ Report FIDE/95/125, 1995

109. Morrison R, Brown AL, Connor RCH, Dearle A. The Napier88 Reference Manual. Universities of Glasgow and St Andrews Report PPRR-77-89, 1989

110. Farkas AM. ABERDEEN: A Browser allowing intERactive DEclarations and Expressions in Napier88. University of Adelaide, 1991

111. Kirby GNC, Cutts QI, Connor RCH, Dearle A, Morrison R. Programmers' Guide to the Napier88 Standard Library, Edition 2.1. University of St Andrews, 1992