This paper should be referenced as:

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Atkinson, M.P. “The Napier Type System”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D. (ed), Springer-Verlag (1989) pp 3-18.

The Napier Type System

R.Morrison, A.Brown, R.Carrick, R.Connor & A.Dearle
University of St Andrews

M.P. Atkinson
University of Glasgow

ABSTRACT

Persistent programming is concerned with the construction of large and long lived systems of data. In designing and building persistent object systems, we are attempting to regularise the activities that are performed on data by programming languages, operating systems, database management systems and file systems. We have identified the following areas of research which we are investigating in the context of persistent systems. They are: controlling complexity, protection of data, orthogonal persistence, controlled system evolution and concurrent computation.

In this paper, we describe the data modelling facilities of the Napier type system. We also demonstrate the flexible and incremental nature of the type checking mechanism that is required for persistent programming. The type system is central to the nature of the Napier language and we will demonstrate how it has been designed to solve problems in the five areas identified above.

1. INTRODUCTION

In our attempts to design and build a persistent information space architecture (PISA) [2], we have identified a number of problem areas. These require solutions before we can achieve our goal of less complex systems in which all programs are integrated with their environments. They are:

a. Controlling complexity: The complexity of the system must be kept under control, so that developers and users can concentrate on the application. This depends on establishing consistent rules which apply throughout the design of the underlying system and being parsimonious in the introduction of new concepts into the design.

b. Protection of data: Large bodies of data are inherently valuable. It is necessary to protect them from misuse and from hardware and software failure. This implies sophisticated type, protection and recovery mechanisms to limit the losses due to component failure.

c. Orthogonal persistence: The discontinuity between the use of short term data, manipulated by program, and long term data, manipulated by the file system or DBMS, causes unnecessary complexity. We have defined the persistence of data to be the length of time for which the data exists and is usable[1]. We aspire to systems where the use of data is independent of its persistence.

d. Controlled system evolution: The uses of data (including program) are neither limited nor predictable. It is therefore necessary to support the construction of unanticipated software systems or databases which make use of existing data (or program) even when the data and program were defined independently of one another. For large scale, widely used or continuously used systems any alteration to part of the system should not require total rebuilding. We require a mechanism which will allow the programmer to control the units of reconstruction.

e. Concurrent computation: A large body of data requires a community effort for its construction and maintenance. Any useful body of data is likely to be of concurrent interest to many users, probably in dispersed geographic locations. Different models of concurrency and transactions may have to be accommodated by the underlying mechanism.

Based on our experience with PS-algol [31], we have designed and built the language Napier [27] to provide solutions in the above areas. The central design theme in Napier is that of the type system and the type checking mechanism which allows flexible and incremental typing and binding within the persistent environment. It is through the type system that the solutions to the above problem areas are resolved. We will discuss each in turn.

2. CONTROLLING COMPLEXITY

Type systems provide two separate facilities that often interact. They are the ability to structure data in a regular form and the ability to protect data from accidental or deliberate misuse. Historically, the domination of the protection aspects of type over the modelling aspects has led developers of large systems to regard type systems as too inflexible for their needs. We believe that it is now appropriate to alter this balance in order to demonstrate the modelling benefits of strong type systems. This we will do in the following sections.

Ideally, we would like a set of types and a type algebra, so that by a succession of operations of the algebra and the provision of parameters, we could define a data type equivalent to any data model or conceptual data model [2]. This we call *the type alchemist's dream* and, as yet, has not been fully achieved.

The complexity of any language or system is determined by the number of defining rules. When applied to type systems this means the model of type, and the rules used to ensure its consistency. There are many models of type and sets of rules which may be applied to ensure consistency. Here we present our initial preferences.

The Napier type system is based on the notion of types as sets of objects from the value space [7]. These sets may be built in, like integer, or they may be formed by using one of the built in type constructors, like **structure**. The constructors obey the Principle of Data Type Completeness [23,24] which states that all data objects have first class citizenship in the language. That is, where a type may be used in a constructor, any type is legal

without exception. This has two benefits. Firstly, since all the rules are general, it allows a very rich type system to be described using a small number of defining rules. This reduces the complexity of the defining rules. The second benefit is that the type constructors are as powerful as they can be since there is no restriction to their domain. This increases the power of the language.

The choice of base types and constructors determines the universe of discourse of a language. This choice is constrained by the data modelling capabilities that are required for the application domain. Below we present the Napier choices that we contend are desirable for persistent programming. It should be remembered that other sets of constructors are possible. The ability to control complexity depends on the method of definition in the first place and on the suitability of the constructors to the application domain in the second.

There are an infinite number of data types in Napier defined recursively by the following rules:

1. The scalar data types are integer, real, boolean, string, pixel, picture, file and null.

2. The type image is the type of an object consisting of a rectangular matrix of pixels.

3. For any data type t, *t is the type of a vector with elements of type t.

4. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, **structure**(I_1:t_1,...,I_n:t_n) is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n.

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

6. For any data types $t_1,...,t_n$ and t, **proc**($t_1,...,t_n$ -> t) is the type of a procedure with parameter types t_i, for i = 1..n and result type t. The type of a resultless procedure is **proc**($t_1,...,t_n$).

7. For any procedure type, **proc**($t_1,...,t_n$ -> t) and type identifiers $T_1,...,T_m$, **proc**[$T_1,...,T_m$] ($t_1,...,t_n$ -> t) is the type **proc**($t_1,...,t_n$ -> t) universally quantified by types $T_1,...,T_m$. These are polymorphic procedures.

8. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, **env** is the type of an environment with bindings of the identifiers I_i to the corresponding types t_i, for i = 1..n.

9. For any type identifiers $W_1,...,W_m$, identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, **abstype** [$W_1,...,W_m$] (I_1: t_1,...,I_n: t_n), is the type of an existentially quantified data type. These are abstract data types.

10. The type **any** is the infinite union of all types.

11. For any user-constructed data type t and type identifiers, $T_1,...,T_n$, t[$T_1,...,T_n$] is the type t parameterised by $T_1,...,T_n$.

12. For identifiers $I_1,...,I_n$ and procedure types $t_1,..t_n$, **process**(I_1: t_1,...,I_n: t_n) is the type of a process with entries I_i and corresponding types t_i, for i = 1..n.

The world of Napier data objects, its universe of discourse, is defined by the closure of rules 1 and 2, under the recursive application of rules 3 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, processes as abstractions over flow control, 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.

In the above, there are two elements in the type system for controlling complexity. The first is in the manner in which the type constructors are defined together with the application of the Principle of Data Type Completeness. With these we get less complex, more powerful systems. The second element in controlling complexity is in the abstraction mechanisms provided by the type constructors. The abstract forms increase the power of the system allowing the complexity of the application to be more readily captured.

3. PROTECTION OF DATA

In determining an appropriate type system for persistent programming, the language designer is faced with a balance between flexibility and safety. The safety in the system is derived from being able to express (or even prove) something about the program before it runs (i.e. statically), in order to improve confidence in its correctness. This explains the wish by most language designers to employ static typing as one of the devices for static checking.

A second aspect of static checking is that the programs so checked are more efficient. By performing the checking statically, the need for dynamic checking is removed, making the run-time representation of the program execute faster and in less space.

Taken to the extreme, totally static systems are not very useful for programming in the large since they cannot accommodate change, the very aspect we frequently wish to model. The mechanism is, however, very safe.

The flexibility in a type system is determined by how late the checking can be delayed before it is performed. In some systems, the type checking is performed dynamically [18]. Reasoning about such systems is more difficult than statically checked systems but they are more flexible. We have already established elsewhere [5,26] that dynamic checking is necessary in providing the persistence abstraction over separately prepared program and data.

Type systems can be devised that offer a mixture of static and dynamic typing more suitable to the requirements of persistent programming applications. Where appropriate, static checking may be used for safety and dynamic checking for flexibility. The mixture of static and dynamic checking is determined by the nature of the particular persistent programming application.

The Napier type system allows the user, by choosing the appropriate constructor, to determine whether the type checking is static or dynamic. The system employs eager type checking where types will be checked as early as possible in the life cycle. This can mean from compile time to run time. The type constructors **env** and **any** are dynamically

checked as are projections out of variants; all other type checking is static. We will illustrate this mixture of static and dynamic type checking by examples of the type abstractions for polymorphism and abstract data types along with type **any**.

Polymorphism is a mechanism whereby we can abstract over type. In Napier,

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

defines the identity procedure for integers. We may also define one for reals, such as

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

Polymorphism allows us to combine the two definitions by abstracting over the type, and creating a generic identity procedure. This may be performed by writing

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

which is the identity procedure for all types; in fact, it has the type ∀t.t -> t . To call the procedure, we may write,

id [**int**] (3)

which will return the value 3. The important point about this form of type polymorphism is that it is all statically checkable. Objects of these types may be kept in the persistent store and then used to generate specialised procedures. This is an important method of software reuse [25].

Abstract data types may be used where we wish the data object to display some abstract behaviour independent of representation type. Thus it is a second mechanism for abstracting over type.

In Napier,

type TEST **is abstype** [I] (value : I ; operation : **proc** (I -> I))

declares the type *TEST* as abstract. The type identifiers that are enclosed in the square brackets are called the witness type identifiers and are the types that are abstracted over.

The abstract data type interface is declared between the round brackets. In the above case, the type has two elements, a field *value* with type *I* and a procedure *operation* with type **proc** (*I* -> *I*). It should be obvious that an infinite number of instances of objects of type *TEST*, each with a different concrete type, may be constructed in Napier.

A comparison can be made with polymorphic procedures which have universally quantified types. These abstract types are existentially quantified and constitute infinite unions over types [22]. Thus *TEST* has type ∃I.abstype (value : I ; operation : proc (I -> I)).

To create an abstract data object we may write,

let inc_int = **proc** (a : **int** -> **int**) ; a + 1
let inc_real = **proc** (b : **real** -> **real**) ; b + 1.0
let this = TEST [**int**] (3, inc_int)

which declares the abstract data object *this* from the type definition *TEST*, the concrete (as opposed to abstract) witness type **int**, the integer 3 and procedure *inc_int*.

Once we have created the abstract data object, we can never again tell how it was constructed. Thus *this* has type,

abstype [I] (value : I ; operation : **proc** (I -> I))

and we can never discover that the witness type is integer.

let that = TEST [**int**] (-42, inc_int)

creates another abstract data object. Although it is constructed using the same real witness type, this information is abstracted over. *this* and *that* have the same type, namely,

abstype [I] (value : I ; operation : **proc** (I -> I))

as does *also* below

let also = TEST [**real**] (-41.99999, inc_real)

We can form a vector of the objects by writing,

let abs_TEST_vec = **vector** @1 **of** [this, that, also]

since they have the same type.

Two abstract types are equivalent if they have the same number of witness types and the same set of identifier type pairs where the witness types may be substituted consistently.

Since we do not know anything about the internal representation of an abstract data object, it is inappropriate to mix operations from one with another. That is, the abstract data object is totally enclosed and may only be used with its own operations.

A second requirement in our system is that we wish the type checking on the use of these objects to be static.

To achieve the above aims we introduce the **use** clause which defines a constant binding for the abstract data object. This constant binding can then be indexed to refer to the values in a manner that is statically checkable.

For example,

```
use abs_TEST_vec (1) as X in
begin
    X (value) := X (operation) (X (value) )
end
```

will apply the procedure *operation* to the value *value,* storing the result in *value,* for the abstract data object refered to by *abs_TEST_vec (1)*. *X* is declared as a constant initialised to *abs_TEST_vec (1)*. Only fields of *X* may be indexed or combined. The scope of the identifiers in the interface is restricted to the brackets following the constant binding identifier.

This could be generalised to a procedure to act on any of the elements of the vector. For example

```
let increment =     proc (this_one : TEST)
                    use this_one as X in
                    begin
                        X (value) := X (operation) (X (value) )
                    end

let lower = lwb [TEST] (abs_TEST_vec)
let upper = upb [TEST] (abs_TEST_vec)

for i = lower to upper do increment (abs_TEST_vec (i) )
```

In the **use** clause the witness types may be named. We may, for example, wish to write procedures over these witness types. This may be written as,

```
use this as X [B] in
begin
    let id = proc [t] (x : t -> t) ; x
    let one := X (value)
    one := id [B] (one)
end
```

which renames the witness type as *B* and allows it to be used as a type identifier within the **use** clause.

Both of the above methods of type quantification can be checked statically. Universal quantification yields a very powerful tool for software reuse whereas existential quantification is a powerful tool in data protection in that the user may not interfere with the representation of the objects of these abstract types.

A second form of type polymorphism employing dynamic checking occurs with the type any, which is the union of all types. Values may be injected into and projected from the type dynamically. An example of the polymorphic identity procedure for type any is

```
let id = proc( x : any -> any) ; x
```

the type of which is **proc** (**any** -> **any**). To call this procedure, the parameter must be of type any. This may be performed by injection, as in the following case:

```
let this_three = id (any (3) )
```

To retrieve the integer value from the object, we must project. An example of projection is the following:

```
project this_three as Y onto
int   : Y + Y     .       .
default : ...
```

The constant binding, to *Y* in the above, is introduced to ensure that no side effect can occur to the projected object within the **project** clause.

This style of polymorphism is no less secure than the universal quantification. The difference is that it is dynamically checked.

The Napier persistent object store is type secure. However, even though Napier provides a rich set of binding mechanisms, there are some activities that are forbidden to ensure the integrity of the type-secure store.

One forbidden activity is type coercion. That is, the system does not allow the types of objects to be arbitrarily changed. Thus, for example, if a string representing the definition of a procedure is to be converted into a procedure, some special mechanism must be made available to check the validity of the conversion. Such a mechanism is called a compiler, which must be available at run-time within the persistent environment if the above objective is to be met.

The Lisp eval function [18] provides such a facility and we believe that some of the attraction of Lisp is that it allows programs to be constructed, compiled and executed dynamically. This is necessary for handling adaptive data, such as programs that produce programs.

The Napier system provides a callable compiler. It is specified by the declaration

let compiler = **proc** (Source : source -> **any**)

Thus, given a textual representation of the source it returns a compiled procedure injected into the type any. To obtain the actual procedure, we must project out of the union. The compiler guarantees the type integrity of the system and is the only program allowed to perform these coercions. We say that it contains the *type magic* for the system [4]. By utilizing the callable compiler, we can produce compiled code for new activities. This compiled code gives us the efficiency that we require and avoids having to have intermediate levels of interpretation on the data to achieve the same functionality.

A final aspect for data security, in Napier, is that all locations in the system may be designated as constant. Since the initial value of the constant may be calculated at run time we call such objects dynamic constants [14]. The constancy applies to all locations and so it is possible to set up complex data structures with constant fields. As applicative programmers know, only a few locations in the store require to be updated in most applications. This can be reflected in Napier by using the dynamic constants.

Thus, for data protection, the Napier type system supports both static and dynamic checking, a type secure object store with a callable compiler and a mechanism for making locations constant.

4. ORTHOGONAL PERSISTENCE

In accordance with the concept of orthogonal persistence, all data objects in Napier may persist. For each incarnation of the Napier object store there is a root of persistence which can be obtained by calling the only predefined procedure in the system, *PS*. Objects that persist beyond the activation of the unit that created them are those which the user has arranged to be reachable from the root of persistence. To determine this the system computes the transitive closure of all objects starting from this root.

The distinguished root of the persistent store graph *PS* is of type environment [9]. Objects of type environment are collections of bindings, that is name-value pairs. They belong to the infinite union of all labelled cross products of typed name-value pairs. They differ

from structures by the fact that we can dynamically add bindings to objects . This is perhaps best shown by example. We will create an environment that contains a counter and two procedures, one to increment the counter and one to decrement it. This may be done by

```
let e = environment ()              !standard procedure to creat an environment
in e let count := 0
!We have now placed the binding count : int := 0 in the environment e.
use e with count : int in
begin
    in e let add = proc (n : int -> int) ; {count := count + n ; count}
    in e let subtract = proc (n : int -> int) ; {count := count - n ; count}
end
!The environment now has three bindings, count, add and subtract.
```

The **use** clause binds an environment and its field names to the clause following the **in**. In the above the name *count* is available in the block as if it had been declared in the immediate enclosing block. The binding occurs at run time since in general the environment value, *e*, may be any expression evaluating to an environment. The binding is therefore dynamic and is similar to projection out of a union. The difference is that here we only require a partial match on the fields and other fields not mentioned in the **use** clause are invisible in the qualified clause and may not be used.

The environment mechanism provides a contextual naming scheme that can be composed dynamically. The **use** clauses can be nested and the environments involved calculated dynamically and therefore the name bindings can be constructed dynamically. This does not yield full dynamic scoping in the Lisp sense since all the objects in the individual environments are statically bound.

5. EVOLUTION OF DATA

Since the uses of data cannot be predicted, it is necessary to provide mechanisms that will support the evolution of data.

We can utilize the callable compiler to implement adaptive data [29]. That is, data that adapts to the needs of the application. This is done by a technique that we call *browser technology* [8]. A browser, in this sense, contains a knowledge base of actions that are selected by certain conditions.

When a browser encounters an object, it interrogates its rule base to see if it has an action for that object. In the most primitive case, the interrogation of the rule base can be a matter of matching on structural type and the action would be the application of the known browser procedure to that object.

If, however, the browser does not find a rule for the object, it must produce one. This can be done by automatically generating the source of a browsing procedure for that type of object. The source is then compiled by the callable compiler and the resultant procedure, keyed by the type, is added to the rule base. The browsing can now continue. Since the browser program itself is data, by virtue of the provision of first class procedures, we have outlined a programming paradigm for adaptive data.

This technique of adaptive data is especially powerful when combined with environments. Since all environments have the same type one may be evolved to another,

with more or less bindings, by assignment. Thus, using the above technique, data may be evolved *in situ* in the persistent store.

6. CONCURRENT COMPUTATION

One of the major motivations for concurrent activity, be it user or machine, is execution speed. The need for concurrency increases as machines approach their theoretical speed limit at the same time as the complexity of the applications increases to require even greater power.

There is, however, a second major motivation for concurrency. Many of the activities that we wish to model are inherently parallel and if we wish to capture the essential nature of these real world activities then language primitives powerful enough to model them are required. One of the major breakthroughs in the design and understanding of operating systems was gained by modelling the system as a set of co-operating sequential processes[10,11]. Since most of the early operating systems modelled in this manner ran on uni-processor machines this modelling was not done to increase speed. It was performed to simplify the complexity of the system being built in order to gain greater insight into its operation. This process-oriented method of modelling, first applied to operating systems, has now been applied to database systems, graphics systems and general problems in computer science[15].

The model of concurrency in Napier is based on CSP [15] and Ada [16]. Process is a type in the language. The type defines the process interface that the process presents to the world. The interface consists of a set of named procedures called entries. External to the process, the entries act like first class procedures. Inside the process the entries do not act like procedures but are used to establish rendezvous with calling processes.

Within a process, the entry name is used to establish a rendezvous. The **receive** clause uses the entry name, its formal parameter list and a clause to be executed during the rendezvous. When a rendezvous is established the body of the particular entry is executed. The caller is suspended from the time the call is made until the completion of the rendezvous. If, however, a **receive** clause is executed before its entry has been called then the callee is suspended until the entry is called. Each **receive** clause defines a body for the entry allowing entry names to have many bodies in a process.

Processes are created and activated by associating a process body (a void clause) with a process type. To implement the rendezvous there is a separate queue of waiting processes for every entry. These queues are serviced in a first come first served basis.

Non-determinism in the system is provided by the **select** clause. This allows one of a number of clauses to be executed. Each clause may be protected by a boolean clause called a guard. To execute the **select** clause all the boolean guards are evaluated in order. An option is open if it does not contain a boolean guard or the boolean guard is **true**, otherwise it is closed. One of the open options is chosen for execution non-deterministically subject to the constraint that if the clause to be executed is an entry clause it will only be chosen if the entry can be received immediately. If none of the options can be immediately executed, e.g. if there is no entry pending, the process waits until one can be.

An example of a generalised index from any ordered type to any type, modelled as a process is given in Appendix I. For the present, we will use on the problem of the 5 dining philosophers [11] and model it using process objects. Before that, however, we must introduce a form of vector initialisation in Napier.

```
let square = proc (i : int -> int) ; i * i
```

defines a procedure that will return the square of its parameter value. A vector initialised by

```
let squares = vector 1 to 10 using square
```

would consist of ten elements indexed from 1 to 10. The elements themselves are initialised by calling the procedure with the index value and using the result as the element value. In this case each element will have a value that is the square of its index.

The solution to the dining philosophers problem is similar to the one proposed by Hoare [15]. In the system there are three types of objects, forks, philosophers and a room in which they dine. Each philosopher sits at a particular unique seat.

Forks have a very simple existence, being picked up and put down in order by one of two philosophers. The five forks modelled as a vector of processes can be defined by

```
type fork is process (pickup, putdown : proc () )

let fork_generator = proc (i : int -> fork)
    fork with
        while true do
        begin
            receive pickup () do {}
            receive putdown () do {}
        end

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

Thus, we now have five processes, *forks (0) .. forks (4)* executing in parallel. Notice that the forks will receive messages from anyone and it is therefore up to the philosophers not to abuse this.

The room has an equally simple existence. Philosophers may enter the room to eat and leave after eating. To avoid deadlock, but not starvation, at most four philosophers are allowed in the room at any one time. The room may be modelled by

```
type Room is process (enter, exit : proc () )

let room = Room with
    begin
        let occupancy := 0
        while true do
        begin
            select
            occupancy < 4   : receive enter () do occupancy := occupancy + 1
                            : receive exit () do occupancy := occupancy - 1
            selected
        end
    end
```

Philosophers enter the room, pick up the left hand and then the right hand fork, eat, put down the forks and leave the room. We must model each philosopher so that the philosopher picks up and puts down the correct forks only. The following will do this

```
type philosopher is process ()

let philosopher_generator = proc (i : int -> philosopher)
    philosopher with
        while true do
        begin
            ! Think
            room (enter) ()
            forks (i, pickup) () ; forks ((i + 1) rem 5, pickup) ()
            ! Eat
            forks (i, putdown) () ; forks ((i + 1) rem 5, putdown) ()
            room (exit) ()
        end

let philosophers = vector 0 to 4 using philosopher_generator
```

Notice that within the closure of each philosopher there is the integer *i* which is in effect the identity of the process and ensures that the correct forks are selected. Putting the fragments together yields the total solution.

It is possible to store data in the form of a process object in the persistent store. This gives a form of object-oriented database where the objects are processes that receive and send messages [28]. More traditional databases are concerned with the stability of data, for integrity, and transaction mechanisms for safe concurrent use.

For stability we provide a low level primitive *stabilise* that ensures that data is copied onto a stable medium. However, we subscribe to the view that it is premature to build mechanisms for atomic transactions into the low level stable store [1,12,13,17]. The stable store provides a very primitive form of transaction that allows the system to recover from unexpected errors, be they hardware or software. Thus it provides primitive (one level) recovery but not atomicity or general (multi process) reversability. There is little agreement on an appropriate generalised transaction mechanism and for the present we deem it safer to build sophisticated mechanisms for atomic transactions, at a higher level of abstraction, using the concurrency and stability primitives.

Users of the persistent information space access data via concurrency and transaction protocols. This is done by encapsulating the data in concurrency and/or transaction abstractions.

7. FUTURE PLANS

There are three major areas in which we would like to develop the Napier type system. They are:

1. Inheritance,
2. Bulk Data Types, and,
3. Type inference.

There are many models of inheritence for database systems. Our particular preference is the multiple inheritance scheme of Cardelli [6]. We do not regard fully blown, unfettered

inheritance as desirable but intend to experiment with bounded universal quantification to achieve our desired aims.

One particularly interesting combination is with processes. The entry list for a process specifies its type and can be considered as the protocol through which it may be accessed. By utilising the multiple inheritance scheme of Cardelli, we can place process types in the type lattice and define a partial ordering of processes. Thus it is possible to define procedures that will operate on processes with at least a given defined protocol. If the process has a more specialised type then that may also be used. For example

```
type shared_int_object is process (write : proc (int), read : proc ( -> int) )
type read_shared_int_object is process (read : proc ( -> int) )

let Read_object = proc [t ≤ read_shared_int_object] (A : t -> int)
                    A (read) (value)

let ron = read_shared_int_object with ...
      ! create a process of type read_shared_int_object

let this = Read_object [ read_shared_int_object ] (ron)
      ! pass it to the procedure Read_object

let fred = shared_int_object with ... ; Write_object [shared_int_object ] (fred)
```

The procedure *Read_object* takes as a parameter an object of type *t* which is a process with at least the entry *read*. In the example, the procedure is called twice with a process parameter. The first *ron* has exactly the entry *read* whereas the second *fred* has more. Inside the procedure, only the entry *read* may be used. By using this bounded universal quantification we can abstract over entry protocols that are common to processes.

Such a mechanism is important in object-oriented programming. Although the mechanism is available for all data types in Napier it is particularly important for processes since they are the main system construction type for self contained objects. By structuring the system into co-operating processes and using the other type constructors as data abstraction mechanisms we can impose an object-oriented methodology.

The inheritance mechanism is also important for controlling change in the system. For example, a process may be changed to give more entries in its interface without altering the procedures that work on at least the supertype. This allows dynamic change to the system subject to the supertype constraints.

We expect persistent programming languages to share some of the applications domain as database programming languages. The success of the latter is based on their ability to capture the regularity of large sets of data - bulk data. One of the earliest experiments was Pascal/R where first order relations were superimposed as persistent objects in a Pascal framework [30]. For persistent programming languages to be successful in this application domain they must also address the problem of large sets of regularly structured data - bulk data.

There appears to be two solutions to this. The first is to identify a bulk data constructor, such as relation or index, define the operations on these and include the constructor in the set of modelling facilities in the language. This was our first approach [3] but so far has proved difficult to integrate with the Principle of Data Type Completeness. The second approach, taken by many object-oriented languages is to introduce a notion of class as a

set of objects of a particular type. Abstract functions can then be defined to operate over all the objects in the class which therefore acts as a bulk constructor.

It is an interesting debate as to whether type inference, as first proposed in ML [20,21], is a desirable property in programming languages. In Napier, some type inferencing is performed in that types in declarations are inferred by the compiler. However, types in abstractions and in certain constructor declarations are required. We believe this to be useful as part of the documentation of the program. However, noise such as instance types for quantification we know to be removable [19].

8. CONCLUSIONS

We have identified the five main areas of research that require investigation before useful persistent programming systems can be constructed. They are controlling complexity, protection of data, orthogonal persistence, controlled system evolution and concurrent computation. Central to our design activity is the construction of a type system for persistent programming and we have discussed how our initial choice for the Napier system may be used to solve some of the problems in the above areas.

For controlling complexity, we have shown how the Principle of Data Type Completeness may be used to define less complex, more powerful type systems. We also contend that our choice of constructors yields a less complex view of the universe for the applications programmer.

For the protection of data, we have provided under user control a mixture of static and dynamic type checking. The type checking can be incremental when we add the method of checking environments. Finally we provide a run time callable compiler to allow for the construction of adaptive data.

For orthogonal persistence, we have proposed a distinguished root of persistence *PS*. *PS* has type environment in the language and may therefore be extended i*n situ*. The dynamic nature of the type checking on environments provides a means for type checking over independently prepared program and data.

For the evolution of data, we have shown how the callable compiler together with environments may be used to provide adapative data within the type secure store.

For concurrent computation, we have provided the data type process together with a non-deterministic **select** clause. We have also speculated on how bounded universal quantification may be used to control the inter-process communication protocols in a type secure object store.

ACKNOWLEDGEMENTS

This work was carried out despite the attempts of the funding bodies to turn us into full time authors of grant applications.

REFERENCES

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

2. Atkinson, M.P., Morrison, R. & Pratten, G.D.

"Designing a persistent information space architecture". *Proc. of the 10th IFIP World Congress*, Dublin, September 1986, pp. 115-120, North-Holland, Amsterdam.

3. Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment". *Proc. of the Appin Workshop on Data Types and Persistence*, Universities of Glasgow and St Andrews, PPRR-16, August 1985, pp. 1-25. In *Data Types and Persistence* (Eds Atkinson, Buneman & Morrison), Springer-Verlag, 1988, pp. 3-20.

4. Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". *Proc. of the 2nd International Workshop on Persistent Object Systems*, Universities of Glasgow and St Andrews PPRR-44, Appin, August 1987, pp. 1-12.

5. Atkinson, M.P., Buneman, O.P. & Morrison, R.
"Binding and Type Checking in Database Programming Languages", *The Computer Journal*. 31, 2, 1988, pp. 99-109.

6. Cardelli, L.
"A semantics of multiple inheritance". In *Lecture Notes in Computer Science*. 173, Springer-Verlag (1984), pp. 51-67.

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

8. Dearle, A. & Brown, A.L.
"Safe browsing in a strongly typed persistent environment". *The Computer Journal*, 31, 2 (1988), pp. 540-545.

9. Dearle, A.
"Environments: a flexible binding mechanism to support system evolution". *Proc. HICSS-22*, Hawaii, January 1989, pp. 46-55.

10. Dijkstra, E.W.
"The structure of THE multiprogramming system". *Comm.ACM* 11, 5, May 1968, pp. 341-346.

11. Dijkstra, E.W.
"Cooperating sequential processes". In *Programming Languages* (editor F. Genuys), Academic Press, London, 1968, pp. 43-112.

12. Fredrich, M. & Older, W.
"HELIX : the architecture of a distributed file system". *Proc. 4th Conf. on Distributed Computer Systems,*. May 1984, pp. 422-431.

13. Gammage, N.D., Kamel, R.F. & Casey, L.M.
"Remote Rendezvous". *Software, Practice & Experience* 17, 10, 1987, pp. 741-755.

14. Gunn, H.I.E. & Morrison, R.

"On the implementation of constants". *Information Processing Letters* 9, 1, 1979, pp. 1-4.

15. Hoare, C.A.R.
"Communicating sequential processes". *Comm.ACM* 21, 8, August 1978, pp. 666-677.

16. Ichbiah et al.,
"The Programming Language Ada Reference Manual". In *Lecture Notes in Computer Science*. 155. Springer-Verlag, 1983.

17. Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment". *Proc. of Data Types and Persistence Workshop*, Appin, August 1985, pp. 86-117. In *Data Types and Persistence* (Eds Atkinson, Buneman & Morrison) Springer-Verlag, 1988, pp. 213-234.

18. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. & Levin, M.I.
"The Lisp Programmers Manual". MIT Press, Cambridge, Massachusetts, 1962.

19. Matthews, D.C.J.
"Poly manual". Technical Report 65, 1985, University of Cambridge, U.K.

20. Milner, R.
"A theory of type polymorphism in programming". *JACM* 26, 4, pp. 792-818.

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

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

23. Morrison, R.
"S-algol Reference Manual". *CS 79/1* University of St Andrews, 1979.

24. Morrison, R.
"On the development of algol". Ph.D. thesis, University of St Andrews, 1979.

25. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C., Dearle, A. & Atkinson, M.P.
"Polymorphism, persistence and software reuse in a strongly typed object-oriented environment". *Software Engineering Journal*, December 1987, pp. 199-204.

26. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"Flexible Incremental Binding in a Persistent Object Store". *ACM.Sigplan Notices*, 23, 4, April 1988, pp. 27-34.

27. Morrison, R., Brown, A.L., Carrick, R., Connor, R. & Dearle, A.
"Napier88 Reference Manual". *PPRR-77-89*. Universities of St Andrews and Glasgow. (1989).

28. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C. & Dearle, A.

"On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments". *Proc. 2nd International Workshop on Object-Oriented Database Systems*, West Germany, 1988. In *Lecture Notes in Computer Science*, 334. Springer-Verlag, September 1988, pp. 334-339.

29. Morrison, R., Dearle, A. & Marlin, C.D.
"Adaptive Data Stores". *Proc. AI'88 Conference*, Adelaide, November 1988, pp. 135-145.

30. Schmidt, J.W.
"Some high level language constructs for data of type relation".*ACM.TODS* 2, 3, 1977, pp. 247-261.

31. PS-algol reference manual. 4th Edition. Universities of Glasgow and St Andrews *PPRR-12*, July 1987.

Appendix I

```
type general_index [KEY, VALUE] is process (
                                Enter           : proc (KEY, VALUE),
                                Lookup      : proc (KEY -> VALUE) )

let generate_general_index = proc [Key, Value] (less_than : proc (Key, Key -> bool) ;
                                fail_value : Value -> general_index [Key, Value])
general_index [Key, Value] with
begin
        rec type index is variant (node : Node ; tip : null)
        & Node is structure (key : Key ; value : Value ; left, right : index)

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

        let i := null_index
                !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, null_index, null_index)) else
        case true of
        less_than (k,i'node (key) )   : { i'node (left) := enter (k, v, i'node (left)) ; i }
        k = i'node (key)              : { i'node (value) := v ; 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 less_than (k, head'node (key) )                              then he
                                                                        else head'node (right)
                if head is node then head'node (value) else fail_value
        end

        while true do
                select
                        : receive Enter (key : Key ; value : Value) do i := enter (key, value, i)
                        : receive Lookup (key : Key -> Value) do lookup (key, i)
                selected
end
```