Napier88 –
A Database Programming Language?

Alan Dearle, Richard Connor, Fred Brown & Ron Morrison

al%uk.ac.st-and.cs@ukc
richard%uk.ac.st-and.cs@ukc
ab%uk.ac.st-and.cs@ukc
ron%uk.ac.st-and.cs@ukc

Department of Computational Science
University of St Andrews
North Haugh
St Andrews
Scotland
KY16 9SS.

Abstract

This is a description of the Napier88 type system based on "A Framework for Comparing Type Systems for Database Programming Languages" by Albano et al.

1 Introduction

This is a description of the Napier88 type system based on "A Framework for Comparing Type Systems for Database Programming Languages" [ADG89]. Napier88 is designed to be a general purpose persistent programming language. The authors envisage the language to be used to construct large, integrated computer systems such as CAD, CASE, hypermedia, database and Office Integration systems. Such systems have the following requirements in common which are met by the Napier88 programming language:

1. the need to support large amounts of dynamically changing, structured data, and,

2. the need for sophisticated MMI.

In order to meet these demands an integrated environment is needed which will support all of the application builders' requirements.

For comparison, it is useful to distinguish between Database Programming languages and Persistent Programming Languages. Database programming languages are not necessarily computationally complete. They often require the use of other ‘conventional’ programming languages [CL88]. More importantly, database programming languages tend to be data centred. This trend has been drawn to its logical conclusion in object oriented database programming languages (OODBPL's) [BBB88,SZ86] where the computational aspects of the system have become attributes of the data.

On the other hand, a persistent programming language is a programming language which treats persistence as an orthogonal property of data [ABC83]. The word "orthogonal" is often conveniently forgotten in many descriptions of programming languages, yet it is the orthogonality that we believe to be important.

Programming languages generally and persistent programming languages specifically are program centred rather than data centred. This means that the programmer tends to write procedures which operate over data rather than treating functions as properties of data. Of course, this does not mean that these languages do not contain any of the ideas of encapsulation recently popularised by Object Oriented languages that have been in the programming language community for many years [LZ74].

2 The Nature of the Type System

The type system of Napier88 has two key roles:

1. as a modelling tool to structure data, and,

2. as a protection mechanism.

The modelling facilities in any language reflect the application areas in which the language designers see their language being used. In Section 3.1 we describe the constructors available in Napier88. The constructors, such as the graphics facilities, reflect the fact that this language is viewed as a tool for constructing large systems such as suites of CAD programs.

It should be noted that the modelling facilities available in Napier88, and indeed available in many other languages, are largely orthogonal to the other language features. We therefore view Napier88 as a shell language on which other experiments such as bulk data constructors for database programming may be constructed. Therefore, the constructors described in Section 3.1 merely highlight the authors' prejudices.

Cardelli and Wegner [CW85] describe a type as a suit of clothes or armour in that it protects an underlying type representation from arbitrary or unintended use. As the designers of Napier88, we very much subscribe to this view of a type. In Napier88, this view of types is, perhaps, more important than in many programming languages due to the longevity of the data being manipulated. The persistent store may contain derived or captured data which is irreplaceable if destroyed or damaged, maliciously or accidentally.

Having a strong type system has many consequences for the design of the store [bro89]. Knowing that an object may not be misused moves protection from the storage architecture into the language architecture. Thus, if a persistent store is only accessed by languages with a single strong type system no store level protection mechanisms [BCC88] are required. This allows the store to be constructed more cheaply and efficiently.

2.1 Does the language have a type system?

We think so.

2.2 Is the language strongly typed? Is type checking static or dynamic?

Napier88 has a strong type system which is mostly statically checked. The Napier language will soon have a compiler that is a first class data object in the store. (However, please note that this is not part of the Napier88 release.) The possibility of dynamically compiling programs blurs the "normal" distinction between static (compile time) and dynamic (run time) type checking. The system employs eager type checking where types are checked as early as possible in the life cycle.

All Napier88 operations are statically checked for type correctness except for projection out of the infinite unions – this operation must always be dynamically checked in all programming languages. A dynamic check must also be made on projection from variants. We will return to the two infinite unions – env and any later in the paper.

2.3 How is type checking used in data definition and for operations?

Type checking is used for both data definition and operations. When types are declared (which they do not have to be – see Section 5.2) the type checker checks the declaration for consistency and "well formedness". All operations are also strictly type checked.

2.4 Is the type system used only for error checking or also for the specification of implementation details?

The type system of Napier88 serves both these roles. We have already mentioned that Napier88 provides the user with a type secure object store. The type security of this store is provided by type checking that is performed over a spectrum of times under user control.

However, the type system may also be used to specify implementation details. For example, we may wish to specify the type of a stack implemented by vectors of integers as:

```
type stack is structure( this   : *int ;
                         pop    : proc( *int -> int ) ;
                         push   : proc( *int,int ) )
```

Alternatively, we may wish to hide the implementation as:

```
type stack is abstype[imp] (  this   : imp
                              pop    : proc( imp -> int )
                              push   : proc( imp,int ) )
```

Abstract types are fully described in Section 5.2.3.

Sometimes Napier88 types are used both to allow type checking and as a specification of implementation details. A good example of this is the vector constructor. In Napier88, a vector is an ordered collection of objects of the same type and as such it is checkable by the type checker. However, a vector is also a store constructor, in this role it definitely specifies implementation details.

3 Expressiveness

3.1 What primitive types and type constructors are available in the language? Are there restrictions on the combinations of type constructors or the form of recursion?

There are an infinite number of data types in Napier88 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 $T_1,...,T_m$. This is the type of a polymorphic procedure.

8. **env** is the type of an environment.

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. This is the type of an Abstract Data Type.

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

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

The world of Napier88 data objects, its universe of discourse, is defined by the closure of rules 1 and 2, under the recursive application of rules 3 to 11.

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

The constructors obey the Principle of Data Type Completeness [mor79a,mor79b] 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 possible since there are no restrictions to their domain. This increases the power of the language.

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 as an bstraction 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.

3.2 What kinds of polymorphism are supported by the type system?

3.2.1 Ad hoc polymorphism

In addition to the parametric polymorphism described below, Napier88 also has some operators which exhibit ad hoc polymorphism. These may be divided into two categories: operators which operate over all types in an ad hoc manner – the universal ad hoc operators; and ad hoc operators which are only applicable to instances of some data types.

The universal ad hoc operators are equality and assignment. Equality and assignment are defined over all types in Napier88 but the code executed is type dependent. Equality always means identity.

The truly ad hoc operations are: the arithmetic operators plus, minus and multiplication; and the comparison operators $<$, $\leq$, $>$ and $\geq$. These are defined over strings, integers and real numbers. No automatic type coercion is performed although procedures, called *float* and *truncate* are provided to convert between integers and real numbers.

3.2.2 Universal polymorphism

Napier88 has only one form of universal polymorphism, namely parametric polymorphism. Napier88 does not have any forms of inclusion polymorphism.

The Napier88 procedure,

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

is quantified by the type *t*. In the language Napier, procedures may be quantified by any number of types, giving the programmer power to abstract over many types.

The parametric polymorphism provided in Napier88 is explicit. That is, quantifiers must always be specified when a polymorphic procedure is defined. Similarly, procedures must be specialised to some concrete type before they are applied. No type inferencing is performed by the type checker. For example, the identity procedure shown above may be specialised to the integer identity procedure by writing,

id[**int**]

or to the string identity procedure, by writing,

id[**string**].

Each specialisation of a polymorphic procedure creates a new procedure instance. The closure created comprises the code for the polymorphic procedure and the new environment. A single instance of a polymorphic procedure may be specialised to many different non polymorphic procedure instances.

In Napier, polymorphic procedures have full civil rights in both their specialised and unspecialised form. Therefore the integer identity procedure, shown above, may be passed as a parameter or returned by a procedure. If it is bound to a location, as follows,

let idint = id[**int**]

then the procedure bound to *idint* has the same functionality as the procedure:

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

The Abstract Data types of Napier88 also give a form of polymorphism in that they are existentially quantified. They are described in Section 5.2.3.

3.3 Infinite Unions

Napier88, provides two infinite unions, namely env and any.

3.3.1 The infinite union "any"

The identity procedure shown above may be written employing dynamic checking rather than static checking, using the type any, which is the union of all types. Values may be injected into and projected from this 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.

3.3.2 The infinite union "environment"

Objects of type environment [dea89] 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 bindings may be dynamically added to environments. Also, all environments are of the same type, namely env. This differs considerably from the statically checked structures of Napier88. Programs similar to the examples using any given above may be written using environments. We will return to environments in section 5.4 when modules are discussed.

4 Types and values

4.1 What are the properties that are possessed by values of all types? Is there a concept of first class values? What are the properties that define first class values? Do all types have these properties?

In Napier88 all values have the same "civil rights"; this is in accordance with the Principle of Data Type Completeness which ensures that all data types may be used in any combination in the language. For example, any data type may be a parameter to or returned from a procedure. In addition to this there are a number of properties that all data types possess that constitute their civil rights in the language and define first class citizenship. All data types in Napier88 have first class citizenship.

The additional civil rights that define first class citizenship are:

1 the right to be declared,

2 the right to be assigned to and to be assigned,

3 the right to have equality defined over them, and

4 the right to persist.

The civil rights of values are of particular interest when persistence is considered. Two rules for persistent data have been given [ABC83]. They are:

> ***The Principle of Persistence Independence****: The persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of program is expressed independently of the persistence of data it manipulates. For example, it should be possible to call a procedure with its parameters sometimes objects with long term persistence and at other times only transient.*

> ***The Principle of Persistent Data Type Completeness****: In line with the principle of data type completeness all data objects should be allowed the full range of persistence.*

In a persistent system the use of all data is independent of its persistence. This notion of persistence may be extended to abstract over all the physical properties of data, for example, where it is kept, how long it is kept and in what form it is kept [AMP86]. The use of the persistence abstraction removes the need to explicitly program for the differences in the use of long and short term data.

Thus, a programming language that supports persistent data objects provides the homogeneous object storage facility required for building many large scale applications and indeed, Atkinson's original motivation [atk78] for investigating persistence was to support Computer Aided Design (CAD) systems.

4.2 Are there values that can be typed that have special properties not shared by all values? What are they and how are they special? Among these might be functions, mutable values and types themselves.

We believe that the mutability of values is not a typing issue. During the execution of a program, the store [str67], or more correctly, locations in the store may be altered. Structured values such as records and arrays may contain locations which may be either mutable or immutable. This belief is reflected in the Napier88 type system which does not refer to mutability or constancy as we like to call it.

In Napier88, all locations in the system may be designated as constant or variable. Since the initial value of constant locations may be calculated at run time we call such objects dynamic constants [GM79]. The constancy applies to all locations and so it is possible to set up complex data structures with constant fields.

It should be noted that, in Napier88, the constancy of a location is not part of the locations' type. Therefore the objects,

```
   type person is structure( name : string ; nearly43 : bool )
```

and

```
   type person is structure( constant name : string ; nearly43 : bool )
```

both have the same type, namely,

```
   structure( name : string ; nearly43 : bool )
```

This decision has consequences in the run time system in that, in the general case, assignments to locations must be dynamically checked for constancy.

4.3 Does the type system allow the construction of objects?

In Napier88 all values are objects in that they have state, identity and operations.

4.4 Is every value an object, or do both objects and non-object values exist? Are objects first class values? What is the exact distinction (if any) between objects and first class values in the language? How do types of objects fit into the type system?

Atkinson and Morrison [AM85] have demonstrated that the ability to support objects with encapsulated state may be modelled by first class procedures. For example, a procedure *unique* may be written in Napier88 that returns a unique integer every time it is called as follows:

```
let unique =                                ! Declaration of the name "unique".
begin                                       ! A block whose value is a procedure.
      let identifier := 0                   ! An encapsulated value.

      proc( -> int )                        ! This procedure is the value of the block.
      begin
            identifier := identifier + 1
            identifier                      ! The procedure returns this value.
      end
end
```

The location *identifier* is encapsulated in the closure of the procedure that is exported as the result of the block. It is this procedure that is bound to the identifier *unique*. This procedure is an object in that it is an entity with encapsulated state and a (trivial) set of operations.

This technique may be extended to return packages of procedures, each with encapsulated state. For example here is the Napier88 definition of an integer stack package, implemented using a vector:

```
type stack is structure( Push : proc( int ) ; Pop : proc( -> int ) )

let myStack =                                 ! Declaration of the name "myStack".
begin                                         ! A block whose value is a package.
        let vec = vector 1 to 100 of 0        ! The stack implementation -
        let top := 1                          ! ignoring errors for simplicity.

        let push = proc( value : int )        ! The push procedure.
        begin
                vec( top ) := value           ! Push new value.
                top := top + 1                ! Post increment top.
        end

        let pop = proc( -> int )
        begin
                top := top - 1                ! Pre decrement top.
                vec( top )                    ! Return top.
        end

        stack( push,pop )                     ! Return the package.
end                                           ! This package is bound to "myStack"
```

In this example two values, namely *vec* and *top*, are encapsulated in the closure of the exported procedures, push and pop. These procedures are returned from the block in an object of type *stack*. Like the earlier example, the instance of stack is an object in that it is an entity with encapsulated state and has a set of operations. This is an example of what Cardelli and Wegner call "first order information hiding". We will return to the subject of information hiding later in section 5.3.

4.5 What is the semantics of the equality relation?

In Napier, the semantics of the equality relation is equality based on identity.

4.6 Is there a concept of null value? Are there different kinds of null values? How do they interact with the equality relation?

Napier88 has a predefined type called null. There is only one value of type null – the literal nil. The only operation permitted on objects of type null is testing for equality,

```
nil = nil,
```

which is tautologous.

Nil values are also provided for pictures and images in the form of the literals *nilpic* and *nilimage*. The value *nilpic* is the picture containing no points, the value *nilimage* is the image containing no pixels.

5 Relationships among Types

5.1 What is the nature of type equivalence?

Type equivalence is based on structural equivalence. In a persistent or distributed environment instances of the same type may originate from many different compilation units. The use of structural equivalence avoids the need for a centralised dictionary.

The key question here is whether there is a central type server or may the schema be distributed over the whole system. In Napier88, (in line with our Scottish Presbyterian roots) we have taken a hard line on this topic and do not require a centralised data dictionary. In fact, in the current implementation, the only place that type information is stored is in the two infinite unions env and any.

5.2 Does the language offer abstract, “concrete” and “unnamed” types?

Napier88 provides abstract, “concrete” and “unnamed” types.

5.2.1 Concrete types

Types may be named, for example,

type person **is structure**(name : **string** ; nearly43 **: bool**)

relates between the name *person* and the type:

structure(name : **string** ; nearly43 : **bool**)

This type declaration introduces a constructor function called person into the local scope. However, the ability to name types in this way is mere syntactic sugar and the instance of the unamed type shown in the next section also has the type *person*. Furthermore, the type *person*, declared above, is equivalent (because of structural equivalence) to the type:

type penguin **is structure**(name : **string** ; nearly43 : **bool**)

In Napier88 it is possible to introduce concrete names for all types in the type domain.

5.2.2 Unnamed types

In Napier88 it is not necessary to declare a type before constructing an object of that type. For example, the programmer may write,

let ron **=** **struct**(name = "Ron" ; nearly43 = **true**)

which creates an structure instance of type,

structure(name : **string** ; nearly43 : **bool**)

bound to the identifier *ron*, without the type ever being named.

There is one exception to this rule and that is abstract data types. The type of an ADT must always be declared before an abstract object is created. This is necessary because the programmer must be able to specify which types are being abstracted over.

5.2.3 Abstract types

Napier88 has a powerful abstract data type (ADT) construct based on the existential types of Mitchell and Plotkin [MP85]. It allows an ADT to be manipulated without being able to discover its implementation or representation. Thus ADT's provide a second mechanism for abstracting over type.

In Napier88, ADT's are described by type. For example, an abstract type may be defined as follows,

```
type number is abstype[ i ]( value     : i ;
                             increment : proc( i -> i ) ;
                             display   : proc( i ) )
```

The type *i* is known as the witness type. Once an abstract type has been created, it is impossible to discover what this type is in reality. The type defined has three fields: a *value* field of the witness type *i*; an *increment* field which is a procedure from *i* to *i*; and a display field which is a procedure that takes an *i* as a parameter.

In an analagous manner to the creation of structures, the name of the type may used to create an instance of an abstract data type. The following example shows the creation of an instance of the type *number*, in this case the witness type is integer.

```
let adt1 = number[ int ]( 1 ;
                          proc( x : int-> int ) ; x + 1 ;
                          proc( x : int ) ; writei( x ) )
```

Similarly, another instance of the same type could be created, with string as the witness type. The implementation shown in the following example uses tabular (base 1) arithmetic.

```
let adt2 = number[ string ]( "1" ;
                             proc( x : string-> string ) ; concat( x, "1" )
                             proc( x : string ) ; writei( length( x ) ) )
```

The two objects denoted by *adt1* and *adt2* both have the same type, that is,

```
abstype[ i ]( value     : i ;
              increment : proc( i -> i ) ;
              display   : proc( i )
```

or *number* for short. They are assignment compatible, may be passed as parameters in place of each other and so on. Furthermore, the two implementations are indistinguishable since they exhibit the same operational semantics.

The types described are truly abstract: they exhibit the same semantics yet have totally different implementations. Obviously, disaster would ensue if a witness type from one implementation were supplied to a procedure from another. Consequently, Napier88 utilises a scoping mechanism to ensure that this may never happen. Witness types may roam free of abstract data types only within a restricted scope.

In the example below, a procedure called *useabs* is defined. It takes an instance of the abstract data type *number* as a parameter. In the body of the procedure, a *value* of witness type is extracted from the abstract type, as are the *display* and *increment* procedures. These are extracted to avoid repeated dereferencing of the object. The procedure *display* is called to show the value of *anumber*. The value *anumber* is then incremented using the *increment* procedure. The value of the procedure is again displayed before the value *incremented* is finally assigned to the location *value* in the abstract type.

```
let useabs = proc( anum : number )
use anum as this in
begin
        let anumber = this( value )
        let display = this( display )
        let increment = this( increment )
        display( anumber )
        let incremented = increment( anumber )
        display( incremented )
        this( value ) := incremented
end
```

The procedure defined above may be supplied with the objects *adt1* or *adt2* as its parameter since these are both of type *number*. The fact that the implementations of *adt1* and *adt2* are different does not matter; it is these implementations that have been abstracted over. In both cases, the first call of *display* will write "1" and the second call will write "2". In both cases, the abstract value "2" is assigned to the field *value* in the abstract type.

The key to the protection mechanism is the renaming that is performed. The object bound to the identifier *adt2* is bound to the identifier *this*. The location denoted by *this* is constant. Furthermore, the scope of *this* is limited to the block associated with the use clause. This is the only way in which abstract objects may be dereferenced ensuring safety.

This mechanism may seem unnecessarily restrictive. However, it is necessary if static type checking is to be performed on abstract data types.

5.3 How are abstract types related to object types?

As stated earlier, Napier88 does not have any syntactic support for objects with hidden state. However, as we have already seen, objects with hidden state may easily be realised using information hiding. In their paper, Cardelli and Wegner [CW85] refer to this technique as *first-order* information hiding. This is in contrast to the information hiding in abstract types where some type information has been abstracted over and hence hidden. Hiding information by the use of existential quantifiers, as in Napier88 abstract types, is called *second-order* information hiding by Cardelli and Wegner. This contrast is illustrated in the example below which shows a point package implemented in Napier88 using first-order and second-order information hiding.

Firstly, using first-order information hiding,

```
rec type Point1 is
structure(  makepoint  : proc( x,y : real -> Point1 ) ;
            x_coord    : proc( -> real ) ;
            y_coord    : proc( -> real ) )

let aPoint1 =
begin
        rec let makepoint = proc( x,y : real -> Point1 )
        begin
                let hidden = struct( x = x ; y = y )
                Point1(  makepoint,
                         proc( -> real ) ; hidden( x ),
                         proc( -> real ) ; hidden( y ) )
        end

        makepoint( 0.0,0.0 )
end
```

and using second-order information hiding,

```
type Point2 is
abstype [point]( makepoint   : proc( x,y : real -> point ) ;
                 x_coord     : proc( point -> real ) ;
                 y_coord     : proc( point -> real ) )

let aPoint2 =
begin
      let make =  proc( x,y : real ->struct( x,y : real )
                  struct( x = x ; y = y )
      let X =  proc( p : struct( x,y : real ) -> real ) ; p( x )
      let Y =  proc( p : struct( x,y : real ) -> real ) ; p( y )

      Point2[struct( x,y : real )]( make,X,Y )
end
```

5.4 Does the language offer a module mechanism and, if so, what are its features? Are modules related to abstract data types? Are modules realised through a type constructor or are they a concept unrelated to the type system?

The language Napier88 does not have any single syntactic module mechanism such as that found in Modula2 [wir82]. Instead, various modularisation mechanisms are provided. We believe that modules and abstract types are orthogonal concepts and are therefore treated orthogonally in Napier88.

Module mechanisms are varied and serve many different purposes, but the two main reasons for providing modules are:

1. to provide a unit of recompilation, and,

2. to provide a unit of system construction.

We will consider each of these requirements in turn. Firstly let us examine the unit of recompilation issue. In Napier88, the smallest unit of recompilation is the sequence. For example, the following is a legal Napier88 program:

```
let a = 7
```

We purport that the unit of recompilation is largely unimportant – but the finer grain the better. What is important is how these units of compilation fit together, or, how a system may be constructed from components. This brings us to the second issue of providing modules as a unit of system construction. We consider this problem to be a binding problem.

The method of composing systems out of components should be simple and, if possible, one of the binding mechanisms already used in the language. This binding mechanism is necessarily dynamic since we wish to construct the system from components in a 'live' system.

Modularisation is realised through the type constructor 'environment'. As we have already stated, 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.

A new (empty) environment may be created by calling the predefined procedure *environment*, which is of type,

```
proc( -> env )
```

For example the programmer may write,

```
let e = environment()
```

which will create a new environment bound to the name *e*. The programmer may then write,

```
in e let a = 7
```

which will create a constant L-value binding and place the binding in *e*. To use the binding the programmer may write

```
use e as a : int in
        writei( a + a )
```

The use clause dynamically binds the name, type, constancy and L-value to the environment expression. If the environment contains at least that name, type, constancy tuple then the binding succeeds and the name is available in the following clause. The binding, which occurs at run time, and is therefore dynamic, is similar to projecting out of a union. The difference here is that we only require a partial match. Other fields not mentioned in the use clause are invisible in the qualified clause and may not be used.

The distinguished root of the persistent store graph is of type environment and may be obtained by calling the predefined procedure called *PS* which also is of type,

```
proc( -> env )
```

The next example creates an environment called *new*, places a vector in it and finally creates a binding to the new environment in the root environment.

```
let new = environment()
in new let avector = vector @1 of [ 1,2,3,4,5 ]
in PS() let e = new
```

We now have an arrangement where *avector* is contained in the environment *new* which is itself contained in the root environment. If the program executing this stabilises the persistent store *avector* and *e* are automatically part of the persistent store, since they are reachable from the root of persistence, and therefore retained. Notice that the name *new* is only used in the local context and is lost when the program terminates whereas the names *e* and *avector* persist.

To illustrate the modularity of environments we will assume that the above program has been executed and that we wish to place an iterator procedure over vectors into the environment *e*. We may achieve this as follows:

```
use PS() as e : env in
          in e let foreach =   proc[t]( v:*t ; op : proc( t ) )
                               for i =  lwb[t]( v ) to upb[t]( v ) do
                                        op( v(i) )
```

This program creates a binding to the procedure called *foreach* in the environment *e*. The procedure is polymorphic and higher order. It applies the procedure *op* to every element of the vector called *v*.

Finally, in yet another program, we may wish to use the *foreach* procedure to sum the vector *avector*. This may be achieved as follows,

```
use PS() as e : env in
          use  e  as  foreach : proc[t]( *t , proc( t ) ) ;
                      avector : *int in
          begin
                  let count := 0
                  let counter = proc( i : int ) ; count := count + i

                  foreach[int]( avector, counter )
          end
```

This method of dynamic composition or binding to environments allows us to compose systems while still retaining static type checking within the compilation unit. The projection performs a one time only dynamic check allowing static checking from that point. It should be noticed that one of the advantages of structural equivalence of type is that two types in different programs may be the same and therefore we can determine type equivalence across program boundaries.

The environment mechanism in Napier88 also provides a contextual naming scheme. The main mechanism for controlling the naming of objects is scope. This can be a very powerful device especially in languages with higher order procedures [AM85]. The use clause introduces names into a context in the same manner as declarations in blocks. Blocks, however, are nested statically whereas the environments can be composed at run time and therefore nested dynamically. This gives a method of binding commonly found in operating systems and suggests that it is best used in that type of activity. That is, when we want to compose objects out of already existing components.

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

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

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

6 Types and subtypes

Napier88 has no notion of subtyping, although we expect to include Cardelli style [car84] subtype inheritance into future Napier releases.

7 Classes and subclasses

7.10 Can objects be removed from classes and subclasses? Can objects be deleted?

In Napier88, objects may never be explicitly deleted. All space (including persistence) management is handled automatically by the underlying architecture.

8 Database Issues

8.1 Is persistence orthogonal to the type system?

Yes, in accordance with the concept of orthogonal persistence, all data objects in Napier88 may persist. For each incarnation of the Napier88 object store there is a root of persistence. 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 with the root.

8.2 What kinds of database schema evolution are supported?

The controlled evolution of data, including program, is of paramount importance in large systems. In particular, persistent systems of data and program must be partially reconstructable and thus incrementally enhanceable. Therefore, persistent systems need to provide binding mechanisms to support incremental development. The understanding of these mechanisms and their significance in the persistent environment is an important consideration in the provision of contextual naming strategies.

The environment mechanism provides a contextual naming scheme that can be composed dynamically. The use clause may be nested and environments involved calculated dynamically or statically, thus permitting the name bindings to be constructed dynamically. This does not yield full dynamic scoping in the Lisp sense, since all objects in the individual environments are statically bound. The technique complements the block structure in the language and completes the context mechanisms required for persistent information spaces.

Environments therefore provide the programmer with a number of binding mechanisms and styles with which to bind objects. Used with care, this mechanism provides the power needed to evolve large flexible persistent object stores containing valuable information.

A technique for schema evolution developed in PS-algol [ps87] may be used within the Napier88 persistent object store.

9 Other issues

9.1 To what extent is type inferencing employed in the language?

The Napier88 compiler trivially infers the types of locations when they are declared and also inferences the types of expressions. The type of any expression may always be inferred statically and therefore types never need to be stated in declarations. For example, the Napier88 programmer may write,

```
let mkpoint = proc( -> structure( x,y : real ) ) ; . . .

let aPoint = mkpoint( 0.0, 0.0 )
```

and the system will infer that the type of mkpoint is

```
proc( -> structure( x,y : real ) )
```

and that the type of aPoint is

```
structure( x,y : real ).
```

9.2 Is there any theory supporting the type system?

There is no underlying theory which allows the programmer to construct proofs over the system. However, the type system is based on the model of types as sets of values.

9.3 Are there implementation factors that are essential to the understanding of the system?

Yes, in order to understand the semantics of the language, the programmer needs to know that all the constructors construct heap objects and that these objects exhibit store (pointer) semantics.

9.4 Are there issues that are not covered in the previous questions that are important to understanding the type system?

We don't think so (but believe there may be).

9.5 What is the status of the implementation of the system?

Napier88 is fully implemented, the system currently runs on Sun workstations. A copy of the system may be obtained by contacting one of the authors of this paper.

10 Conclusions

The Napier88 language was developed as part of the PISA project [AMP86]. The language itself is intended as a testbed for our experiments in type systems, programming enviroments, concurrency, bulk data types and persistence. At the present time we do not have implementations of all our ideas but have constructed the Napier system in such a way that our future experiments may be conducted orthogonally and cheaply.

11 References

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

[ADG89] Albano A., Dearle A., Ghelli G., Marlin C., Morrison R., Orsini R. & Stemple D. "A Framework for Comparing Type Systems for Database Programming Languages". Proc. 2nd International Conference on Database Programming Languages. Oregon (1989).

[AM85] Atkinson M.P. & Morrison R. "Procedures as persistent data objects". ACM.TOPLAS 7,4 (1985).

[AMP86] Atkinson M.P., Morrison R. & Pratten G. "Designing a Persistent Information Space Architecture". In *Information Processing 86* {Proceedings of IFIP 10th World Computer Congress, Dublin, Ireland}, Kugler H.J. (Ed), pp. 115-120 (North-Holland, Amsterdam, 1986).

[atk78] Atkinson M.P. "Programming Languages and Databases". Proc VLDB, pp. 408-419 (1978).

[BBB88] Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lecluse C., Pfeffer P., Richard P. & Valez F. "The Design and Implementation of O_2, an Object Oriented Database System". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. In *Lecture Notes in Computer Science, 334.* Springer-Verlag, pp. 1-22 (September 1988).

[BCC88] Brown A.L., Connor R.C.H., Carrick R., Dearle A. & Morrison R. "The Persistent Abstract Machine Version 4.0". Universities of St Andrews and Glasgow PPRR-59 (1988).

[bro89] Brown A.L. "Persistent Object Stores". Ph.D. thesis, University of St Andrews (1989).

[car84] Cardelli, L."A semantics of multiple inheritance". In *Lecture Notes in Computer Science. 173,* Springer-Verlag, pp. 51-67 (1984).

[CL88] Connors T. & Lyngbaek P. "Providing Uniform Access to Heterogenous Information Bases". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. *In Lecture Notes in Computer Science, 334.* Springer-Verlag, pp. 162-173 (September 1988).

[CW85] Cardelli L. & Wegner P. "On Understanding Types, Data Abstraction and Polymorphism". Computer Surveys, 17, 4, pp. 471-522.(December 1985).

[DB88] Dearle A. & Brown A.L. "Safe Browsing in a Strongly Typed Persistent Environment". The Computer Journal 31,6, pp. 540-545 (December 1988).

[dea89] Dearle, A. "Environments: a flexible binding mechanism to support system evolution".Proc. HICSS-22, Hawaii..Vol II, pp. 46-55 (January 1989).

[GM79] Gunn, H.I.E. & Morrison, R. "On the implementation of constants". Information Processing Letters 9, 1, pp. 1-4 (1979).

[LZ74] Liskov B.H. & Zilles S.N., "Programming with abstract data types", ACM SIGPLAN Notices 9,4 (1974).

[mor79a] Morrison, R. "S-algol Reference Manual". University of St Andrews CS 79/1 (1979).

[mor79b] Morrison, R. "On the development of algol". Ph.D. thesis, University of St Andrews (1979).

[MP85] Mitchell, J.C. & Plotkin, G.D. "Abstract types have existential type". Proc POPL (1985).

[ps87] "The PS-algol Reference Manual fourth edition", Universities of Glasgow and St Andrews PPRR-12 (1987).

[str67] Strachey C. *Fundamental concepts in programming languages*. Oxford University Press (1967).

[SZ86] Skarra A. & Zdonik S.B. "An object server for an object-oriented database system", Proc International Workshop on Object-Oriented Database Systems, Pacific Grove California, pp. 196-204 (September 1986).

[wir82] Wirth N., *Programming in Modula-2*, Springer-Verlag (1982).