This paper should be referenced as:

Atkinson, M.P. & Morrison, R. “Types, Bindings and Parameters in a Persistent Environment”. In **Data Types and Persistence**, Atkinson, M.P., Buneman, O.P. & Morrison, R. (ed), Springer-Verlag (1988) pp 3-20.

Types, Bindings and Parameters in a Persistent Environment

Malcolm P. Atkinson
University of Glasgow, Glasgow, Scotland G12 8QQ.

Ronald Morrison
University of St Andrews, St Andrews, Scotland KY16 9SX.

1. INTRODUCTION

Our experience of persistent data in PS-algol [ABCCM83] and surveys we have conducted [AB85] [ABCCM81] have led us to identify various requirements on the stores embodied in the design of a programming language. Axiomatic in our approach to language design are:

(i). languages should be strictly typed;

(ii). type systems can be discovered that will permit total systems to be built, strictly typed throughout;

(iii). languages should eventually provide convenient and consistent notations for **all** activities inherent in programming.

This leads to a search for constructs that handle activities currently not supported. The requirements identified for persistent data are:

(i). the rights to transience or longevity are completely independent of the type of data;

(ii). the data types should be expressive enough to capture the structure and regularity that will be required of the data;

(iii). the data types should include program, as procedures and abstract data types;

(iv). the data types should include a general purpose indexing mechanism, both polymorphic and variadic;

(v). the data types should include a bulk data type with operations on collections of data;

(vi). the data types should include some form of inheritance to model specialisation;

(vii). the types and bindings used should allow system evolution on an incremental and localised basis;

(viii). there should be mechanisms to permit and control shared and concurrent usage;

(ix). it should be possible to encapsulate any sequence of operations on the store into a transaction, which behaves as a single operation;

(x). there should be privacy and access control mechanisms.

Currently we are engaged in the design of a new language, Napier, to succeed PS-algol. This paper presents some aspects of the design of the binding and type checking we are considering. These particularly address requirements vi and vii above. Requirement vii is equivalent to the requirement for data independence: schema editing without having to recode, recompile or relink all application programs, a requirement identified by the database community in the early 1970s. It is addressed in our design by controlled and localised incremental binding and type checking, in the context of a structure we call name spaces. In compliance with requirement i, we do not constrain its use to long term data, which implies that it may also be used to allow parts of a program to evolve, and may be used to construct flexible interfaces within a program.

We are able to show that these name spaces, structures, records, abstract data types and relations may all be put into a consistent framework described in terms of associating a formal store with an actual store, similar to parameter mechanisms. We also show that this enables, at least in principle, many activities previously dealt with outside the programming language, to be supported from within it.

The requirement of data description, is met by a polymorphic type system, discriminated unions and abstract data types. This is described in detail elsewhere, but shown to be consistent with name spaces in sections 5 and 6 of the paper. In particular, the type matching based on signatures, and the provision of an environment by an ADT are made to have syntactic and semantic similarities with the matching and binding of name spaces. The projection out of a discriminated union to a specific type is shown to have a similar dynamic binding and checking requirement to the dynamic use of name spaces.

The benefits and mechanisms for providing program as part of the persistent type system we have reported elsewhere [AM84].

A relational type constructor, without restrictions on its component types, provides for indexing, bulk data structures, operations and functions stored by enumerating their results. This is briefly described in section 7.

2. BINDING

In programming language systems there are a number of schemes for binding names of objects. Landin [LAND66] proposed the Principle of Correspondence where the methods of binding in any language are the same for both declarations and parameters. More formally Tennent [TENN77] stated that for any formal parameter F (ie the name being declared) and compatible actual parameter A (the value), the effect on the block body of the qualifying declaration F=A should be identical to an abstraction body having F in its formal parameter list when A is the corresponding actual parameter. For elegant languages there should be a one-to-one correspondence between the methods of binding names to objects in declarations and parameters.

This one-to-one correspondence includes methods of parameter passing (call by value, call by reference etc.) and time of binding (static or dynamic). For the present we will concentrate on time of binding.

In static binding the association between name and object can be determined by a static analysis of the program. The duplicate but legal use of names can be resolved by the static scope rules. For example

```
let a := 3 ; let b := 4          ! a contains 3, b contains 4
begin
     let a := 16                 ! a contains 16, b contains 4
     . . .
     b := a * b                  ! a contains 16, b contains 64
     begin
          let b := 93            ! a contains 16, b contains 93
          . . .
          a := a + b             ! a contains 109, b contains 93
     end
end
```

This type of binding was first introduced by algol 60 [NAUR63] and has been continued in the algol tradition by languages such as algol 68 [VANW75], Pascal [WIRT71] and Ada [ICHB83]. It should be noted that this type of binding is conventionally made up of two parts, static scoping and static type checking. A variation is to allow static scoping of names but to check the types of objects dynamically. This method of binding is used in the language SASL[TURN79] and provides a measure of polymorphism.

In dynamic binding the association between name and object is made during the dynamic evaluation of the program. For example

```
let a := 3
let c := proc()
         begin
              . . .
              a := -2
         end
c()                          ! a contains 3 before and -2 after
begin
     let a := -19
     . . .
     c()                     ! a contains -19 before and -2 after
end
```

Dynamic binding is made up of dynamic scoping with either static or dynamic type checking. For static type checking with dynamic scoping all objects with the same name in the program must have the same type. It is much more common to find dynamic scoping linked with dynamic type checking as first provided by LISP [MCAR62]. For the rest of this paper, unless explicitly stated we will use the term static binding to mean static scoping with static type checking and dynamic binding to mean dynamic scoping with dynamic type checking.

The advantage of statically bound programs is that some errors may be detected early in the software life cycle. This is so important to some language designers such as those of ML [MILN84], Argus [LHJSW83] and Pebble [BL84] that they insist that all binding be static. Where it is possible to perform static checking it is also possible to make the programs more efficient since checks can be factored out.

In large scale systems the issue of binding is of major importance and it is in such systems that static binding has hidden costs. Any change to program or data form requires a recompilation to re-establish the bindings. For large systems involving

software evolution or the distribution of software this recompilation may be prohibitive in cost.

It is interesting to note that to overcome these costs and to accommodate dynamic evolution of programs and data, most programming languages allow file names to be bound dynamically and read statements to involve a dynamic type check. This is true even in, so called, strongly typed languages such as Pascal. Indeed it is not well recognised that these languages allow static binding within the program and resort to the operating system, file system or database management system to provide dynamic binding. The two binding mechanisms in these languages almost never obey the Principle of Correspondence completely.

In the context of persistent systems there is a requirement for both types of bindings. A static binding would interpret the above file names in the compilation environment, thus freezing the program and data form. The second type of binding would interpret the file names in the context of the run time environment. This is dynamic binding.

We assert that both methods of binding are necessary for large scale system construction and evolution. We recognise that it is advantageous to statically bind wherever possible with dynamic binding available where necessary. We would expect small objects to be statically bound to form larger objects, with interfaces between those objects being dynamically bound. We coin the term flexible incremental binding (FIB) to describe this mixture of bindings, which we expect to obey the Principle of Correspondence.

3. A MECHANISM FOR CONTROLLED DYNAMIC BINDING

In Section 2 the case for some dynamic binding (and consequently dynamic type checking) was made. In this section we introduce name spaces, an environment mechanism that permits the following:

(i). the storage of bindings in a name space;

(ii). the dynamic use of names from a name space;

(iii). the static use of names from a name space;

(iv). the evolution of the names available in a name space;

(v). safe exchange of arbitrary data between parts of the system - especially between the permanent store and programs.

Remember that a name may be statically or dynamically bound, and the system may determine the type and object associated with the name. However, we wish to compile most code in contexts where there is enough information to statically bind and statically type check. Even when dynamic binding or checking the strictness of type checking will not be compromised.

3.1 Abstract name spaces

A name space expression is written

ns identifier-list **from**

sequence
end from

In such an expression, declarations for each of the identifiers in the identifier-list will appear in the sequence after **from**, giving each identifier a type and an associated object. For example

```
let an1 = proc( x : int ; y : string -> ns )
          ns a,b,c from
             let xx = x * x                ! let is an initialising
             let yy = y ++ y               ! declaration in Napier,
             let a = 3                     ! with type inferred from
             let b:= xx - 2 * x + 5        ! the expression
             rec c = proc(i:int->string)   ! recursive declaration
                     if i<= 0 then yy
                     else c( i - 1 ) ++ c( i - 1 )
          end from
```

All instances of the name space yielded by this procedure start off by containing three triples as shown in the following table

identifier	associated object's type	associated object's value
a	**int**	3
b	**var int**	a location initialised to x^2-2x+5
c	**proc** (**int** -> **string**)	a procedure to yield 2^n doublings of the given string

Locations *a* and *c* are non updatable, ie constant. We show later the similarity between these name spaces and abstract data types, records and structure classes.

3.2 Instances of name spaces

To form an instance of a name space we provide parameters to the procedure.

For example:

```
let n1 = an1( 3,"to" )
```

This would yield a name space which presents the environment

```
{ a : int                      : 3,
  b : var int                  : var containing 8,
  c : proc( int -> string )    : a proc which will give 2^n doublings
                                 of "to" }
```

3.3 Presenting environments

When we write

```
let sequence1 in sequence2
```

in languages such as ML, *sequence2* is compiled in the context of the environment yielded by *sequence1*.

When we write

```
proc( parameters )
       statement
```

the 'statement' here is compiled in the context of an environment specified by 'parameters' but whose values are provided by the 'actual' parameters at the time of the call.

A name space provides an environment in a similar way, and the values provided are those left in the name space when it was created, or when it was last used. There are two constructs, one in which the name space instance used is statically determined and the other where it is dynamically determined.

3.4 Nomenclature

The nomenclature used is analogous to that for procedure parameters. We consider name spaces to be an abstraction of store of arbitrary permanence. The name space information at the time a code sequence is compiled constitutes a formal store and that used at the time it is executed is the actual store. The relationship between the formal and actual store differs in the above two constructs.

3.5 Static invocation of name spaces

A name space may be used to collect values, for example, left by previous runs of this and other programs. If it is known when a program is written which name space will be used, then an expression may be written which always yields the same result, and which can be evaluated both at compile time and run time, to identify that name space. The compiler can then be instructed to compile a sequence of code in the context of that name space's provided environment. For example

```
with n1 compile
     sequence
end compile
```

In the sequence, the identifiers *a*, *b* and *c* will be available with type and associated object as specified in 3.2 above. If, in the sequence, there was the statement

```
b := b + 1
```

then successive uses of this program would find new values in *b* , though *b* is still associated with the same object.

```
with expression compile statement
```

will be shown later to be a general form allowing the environments yielded by ADTs (which also provide records and structures) to be used. Indeed, the expression may also take the form of a void sequence containing declarations, in which case it is a denotation of a static environment in which to evaluate the statement or sequence after **compile**. In

that case, and when a name space is statically bound, the statement after **compile** is always executed using the same actual store determined by the expression before **compile**.

With such static binding to name spaces we can model the stored workspace mechanism, called "persistence", in Poly [MATT85a].

3.6 Dynamic binding of name spaces

A statement of the form

```
using expression with signature compile
      sequence
end compile
```

indicates dynamic binding is required, then a check is made at compile time that the expression will yield a name space. The sequence is compiled with the environment of names and types specified by the signature. Thus the signature specifies a formal store, each time the statement is executed, the expression provides an actual store, possibly different on each occasion. It is therefore necessary to check that the store on this occasion holds data with the names and types specified by the signature. Thus the dynamic binding and checking is localised to the **using** . . . **compile** construct.

3.7 Data evolution

The raison d'être for name spaces is the need for a mechanism to accommodate and localise change.

A name space may be changed by the entry of new bindings and the withdrawal of old ones. For example

```
extend expression with identifier-list from
       sequence
end from
```

requires that expression yields a name space, and updates this name space **in situ**, adding the name: type: object triples identified and initialised by the code from **with** to **end from**. Similarly,

```
drop identifier-list from expression
```

removes all the definitions associated with the names given in the identifier-list from the name space yielded by the expression.

In order that a check which has been applied before a **compile** cannot become invalid during the sequence that follows, a lock is established on a name space for the duration of the sequence which prevents **drop** operations on that name space.

3.8 More about checking

When a name space is used statically it is necessary to check that it has properties required of it. To do this, each name space keeps a count of the **drop** operations applied

to it. At the time of compilation of **with** expression **compile**, the **drop** count value is recorded in the code, at execution time, the current **drop** count is checked to ensure that no significant changes have take place. The compiler has to access the name spaces at compile time to obtain the names and types from it. The **drop** count check ensures the changes to the name space (its address is embedded in the compiled code) have not invalidated the original conditions.

Both dynamic and static binding constructs therefore only check on the names they use, consequently both mechanisms permit a name space to be extended without impact on existing programs.

In the dynamic case, the object bound to a name may change - either because of use of a different name space as the actual store or because of a **drop** followed by an **extend**. It is intended that the reuse of the same name with the same type is an indication that the same meaning and use is implied. Consequently the dynamic binding allows replacement of components, as does assignment to a variable object.

3.9 Assignment and equality

Assignment of name spaces is reference assignment and their equality is an identity test.

3.10 Illustrating the uses of name spaces

The intent of our languages is to accommodate more of the programmer's work within a single language. A few examples will illustrate this.

Two examples of name spaces which do not necessarily involve persistence are presented first. To call the compiler at run time in order to obtain executable results, type definitions etc. presents problems in typing the parameters of the procedure *compile*. With name spaces we can type it

```
proc( env : seq of ns ; source : sourcetype ; results : ns )
```

The sequence of name spaces identifies the context of the compilation, and the results are returned by extending the name space called *results*.

Since any programmer calling *compile* will have conventions on the names used and the types of objects generated, the results of the compilation can be extracted by a dynamic name space binding and subsequently used statically.

In code like the *compile* procedure above or in a general purpose index, data input, query or data output mechanism, the programmer will wish to report to the invoking code errors detected. We assume that these errors will be reported via an exception mechanism which conveys information via parameters. In the case of these general purpose applications the packing of the culprits into a name space allows culprits of any type to be conveyed back to the invoker.

In languages currently in widespread use, dynamic association between data and program is achieved by using names (usually as character strings) to access data in a file catalogue (or directory) maintained by the operating system. A name space may serve in this role, allowing any data to be named externally to the program, but now requiring that it is named and typed consistently.

As a name space is itself an object in the language, a name in a name space may be bound to a name space. Hence hierarchical naming structures similar to directory hierarchies are possible.

Program libraries are obtained by putting procedures or abstract data types in name spaces. The programmer can choose whether to statically bind to a routine in the library and thus find that certain library maintenance operations result in the program needing recompilation, or to bind dynamically so that the program will operate as long as the library provides a routine of the expected name and type. The type checking, binding and loading for all data objects, which have been described and prototyped[AM84], replace such things as: type checking linkers, consolidators, APSE databases and associated utilities. Separate compilation is achieved by running programs which leave program parts in such a program library.

Many operating systems identify a home directory for each user, where a user is identified by a string. This mechanism can be obtained by building a relation (or similar associative structure) which given a string returns a name space. Of course, a name space itself would be an adequate associative data structure if we use names to identify users.

Permanent storage is usually provided by the filing system of an operating system. Programs use this storage to convey results to later programs. Even applicative languages, and those that save workspaces, perform updates on such a store. By distinguishing one or more name spaces, say PermanentStore etc, which for a new system starts empty, and arranging to preserve all data reachable from them, as we have explored in PS-algol [PS85], then permanent storage is provided and the interprogram communication previously provided by the file system may now be coded within the language. Furthermore different users may adopt different naming schemes and organisations of permanent store.

A common requirement is to name a complex data structure, so that an instance of it can be identified for an activity with a given program. For example, a CAD system may be used for many projects. The data associated with each project may be a name space, and all of these may be named in PermanentStore. The current programs constituting the CAD system may be in another name space, say 'CAD', also made accessible by being bound in PermanentStore. The user may apply any program (procedure) from CAD to any of the project databases. The vendor of the CAD system may provide replacement versions from time-to-time, which will be installed, replacing the previous value of 'CAD'. They can be written so as to bind to the existing projects, but so as to extend them to contain new types of data. Bindings via name spaces of customised programs the package user has written will still run. Thus revised software or other data may be delivered to a customer without revealing the program source or proprietary data structures, and this can be used with existing data and software, which may represent a substantial investment by the customer. Such dynamic reassembly is essential with large commercial systems, and is not provided by systems dependent only on static binding, when they claim to provide persistence.

3.11 A comparison with other systems

As has already been remarked, the static binding of name spaces, achieves the same effect as the statically bound work spaces, such as those in Poly. Pascal/R [SCHM77] has a permanent type constructor **database** which provides an environment which is a binding of names to relational values. Only one is permitted per program, and its type is defined as for all types in Pascal, so that the program is compiled in the statically

determined formal store. At execution time the actual store is determined by a parameter to the program, and is, presumably, dynamically checked to be of a compatible type. Transient instances of **database** cannot occur, it cannot be used in other structures and its contents are limited in type.

In Galileo [AOO82] **use** statements add bindings to a standard environment which is permanent. **enter** statements arrange for subsequent environments to be compiled in the identified environment. This is equivalent to writing an

```
extend PermanentStore with name from declaration
```

and writing

```
with PermanentStore compile statement
```

respectively. Galileo does not appear to accommodate any form of dynamic binding, but as environments may be held in structures, and may hold data of any type, it achieves the same power of structured naming as the static name space mechanism we describe does.

In Amber [CARD85], some of the effects of dynamic binding are achieved using inheritance. As the dynamic binding of name spaces only checks the names in the signature the mechanism proposed offers similar inheritance. **extend**ing a name space then becomes equivalent to specialisation.

In Taxis [MW80], all structures are permanent and one statically bound implied name space seems to be the underlying model.

In Adaplex [SFL83], each database is restricted to hold only entities and is statically bound to the programs that use it. It is always persistent. With a type declaration corresponding to each entity type, a name space corresponding to each database, and using only static binding of name spaces, the persistence of Adaplex could be modelled by the name spaces proposed here.

Pebble is also intended to facilitate the assembly of large systems, but is based on the assumption that dynamic binding is not required. The triples stored in a Pebble environment seem to be the same (except for details) as those in our name spaces. However, we have proposed additional operations and an additional form of binding, to permit system evolution in large systems.

3.12 Outstanding problems

With file directories it is possible to write general utilities which scan a directory, often interactively, revealing the directory's contents and possibly allowing user action - e.g. delete - on each file. Some equivalent operations on name spaces may be useful, however, it is difficult to identify a primitive in terms of which they may be written. For example, how does one type the control variable of an iteration over a name space and how does one represent and manipulate names? This problem is not peculiar to name spaces, it also arises in scanning other structures that yield an environment when writing general utilities for browsing, diagnosis, statistics collection and generalised data input and output. Current practice is to step outside the type system or to depend on calling the compiler. Our desire to support more of the total programming activity within the strictly typed language leaves us confronting this problem.

4. TYPES AND DATA STRUCTURES

The data types in Napier are defined by rules of the following form:

a. The primitive data types are integer, real, boolean, picture, pixel, and string;

b. image is the type of an image made up of pixels arranged as a rectangular matrix;

c. For any data type T, *T is the data type of a vector with elements of type T;

d. For any data types T_1,...T_n and T, **proc**(T_1,...T_n->T) is the data type of a procedure taking parameters of type T_1 to T_n and producing a result of type T. The type of a similar resultless procedure is **proc**(T_1,...T_n);

e. For any data types t_1,...t_n,...t_m and identifiers id_{n+1},...id_m, id(t_1,...t_n){id_{n+1}:t_{n+1},...id_m:t_m} is the type of a user defined type with parameters t_1,...t_n and signature {id_{n+1}:t_{n+1},...id_m:t_m} and type name id;

f. For any data types t_1,...t_n, $t_1|t_2|...|t_n$ is the type of a union of the types t_1,...t_n;

g. **ns** is the type of a name space;

h. For any data types t_1,...t_n,...t_m and identifiers id_1,...id_n,...id_m then **rel**(id_1:t_1,...id_n:t_n->id_{n+1}:t_{n+1},...id_m:t_m) is the type of a relation with index (id_1:t_1,...id_n:t_n) and result (id_{n+1}:t_{n+1},...id_m:t_m).

The universe of discourse is defined by the closure of rules (a) and (b) under the recursive application of (c), (d), (e), (f), (g) and (h).

A type in Napier is an abstraction over a declaration just as a function is an abstraction over an expression and a procedure an abstraction over a statement. The main difference between these abstractions in a language with higher order functions is in the use of names. A type definition introduces names into the environment which may then be used to access values of the type. The type name itself may also be used in subsequent declarations. This type mechanism allows concepts such as records, structures and abstract data types to be captured in one. For example, a Pascal record may be modelled by

```
let complex = type( rpart,ipart : real ) is rpart,ipart

let add = proc( a,b : complex -> complex )
          complex( a_rpart + b_rpart,a_ipart + b_ipart )

let a = complex( 3.1,2.1 ; let b = complex( 3.1,-4.1 )

let c = add( a,b )
```

In the above, the type declaration introduces the name *complex* which can be used as a type identifier and as the type constructor. Objects of the type may be created by passing parameters to the constructor. For example

```
complex( 3.0,-42.1 )
```

creates an object of the type *complex* defined above. A value of the type can be obtained using the underbar notation. Thus

```
a_rpart
```

yields the *rpart* value of the complex object *a* . Notice that the representation of the type is not hidden in this example.

Each type has a signature that is used in implementing both polymorphism and name spaces. The above type has the signature {*rpart*: : **real**,*ipart* : **real**}. The type is parameterised in the same manner as procedures.

The mechanism may also be used to implement abstract data types. For example, to hide the representation of complex we could write

```
let compTYPE = type() is add,constr from
    let complex = type( rpart,ipart : real ) is rpart,ipart
    let add = proc( a,b : complex -> complex )
              complex( a_rpart + b_rpart,a_ipart + b_ipart )
    let constr = proc( a,b : real -> complex )
                 complex( a,b )
    end from

let mycomplex = compTYPE()
let add = mycomplex_add ; let create = mycomplex_constr
let a = create( 3.0,2.0 ) ; let b = create( 3.1,-4.2 )
let c = add( a,b )
```

Using the normal algol scope rules, the definition and therefore the selectors on *complex* are hidden to the outside world. Only the primitive functions *add* and *constr* are exported.

4.1 Type matching

One major problem that arises in any type system is the meaning of the equality of types. This problem is made more difficult when the language allows types to be stored and reused as is the case in Napier. As an example of these difficulties, we may write

```
let newtype = if . . . then one type definition
              else another type definition
```

That is, the *if* clause may return a type, as may a procedure or a type itself and we cannot determine which one at compile time. Russell and Poly have different solutions to this problem and Napier proposes a third. A type is a data object in Napier which may therefore be named and stored like all other data objects. A type has a parameter list which is used to create objects of the type and a signature which specifies the set of values for the type. Two expressions which yield types are only type compatible if the types have the same parameter list in one to one correspondence and the same signature. Thus the above *if* expression is only valid if the two type expressions are compatible.

Type names may be declared and used as a shorthand in parameter lists and signatures. This is the most convenient way of using types but raises other problems in type checking in languages with higher order procedures. We would like objects with the same type name to have the same type. However consider the following program

```
let c = begin
            let complex = type( b : int ) is b
            let knit = proc( a : complex )
```

```
                    . . .
                knit
            end
    let ccomplex = type( rpart,ipart : real ) is rpart,ipart
    let a = complex( 1.5,-2.3 )
    c( a )          ! this is an illegal call of c
```

Clearly the call of *c* is illegal since it is really the procedure *knit* which refers to a different *complex* . To overcome this problem, two objects only have the same type if the types have the same name, the parameter lists are in one to one correspondence and they have the same signature. This allows types to have different internal properties but still be regarded as the same. The significance of this is clear to anyone who has tried to evolve persistent data using both old and new versions simultaneously.

In Napier, there is a union of all types called **type**. This is used when storing types in a name space or passing them as a parameter. All types match the type **type**. However as we will see the type may be restricted by a signature to allow a more specific partial match for a particular type.

5. DISCRIMINATED UNIONS

To define recursive data structures we use a discriminated union. For example to build a list of integers we may use

```
    rec intlist = union type() |
                  type( hd : int ; tl : intlist ) is hd,tl
```

The list may be made up of pairs or an empty value. The parameter lists of the constituents of the union must be unique in order to distinguish them on initialisation. Alternatively they may be named. To construct and print a list we could use

```
    let nil = intlist()

    let printlist = proc( head : intlist )
                    while head ~= nil do
                    begin
                         write head_hd
                         head := head_tl
                    end

    let head := intlist( 1,nil ) ; head := intlist( 2,head )
    printlist( head )
```

It is interesting to note that projection from a union always carries a run time check. In the above case we must always check that the selector is appropriate to the object. The check can be factored by a project operation. For example

```
rec scalar   = type( name : string ) is name &
    cons     = union type() |
               type( hd : comptype ; tl : cons ) is hd,tl
    PROC     = type( args : cons ; t : comptype ) is args,result &
    comptype = union scalar | cons | PROC

let nil = cons()

let eq = proc( a,b : comptype -> bool )
```

```
project
a onto scalar and b onto scalar : a_name = b_name
a onto PROC and b onto PROC : eq( a_args,b_args ) and
                              eq( a_result,b_result )
a onto cons and b onto cons : ( a = nil and b = nil ) or
                              a ~= nil and b ~= nil and
                              eq( a_hd,b_hd ) and
                              eq( a_tl,b_tl ) )
default                     : false
end project
```

In the above *scalar* and *PROC* case the type checks can be factored out at compile time. However the other projection only projects onto another union and a further projection is required to factor out all of the checks. The patterns of ML could be used here.

6. POLYMORPHISM

The method of defining polymorphic functions in Napier is taken directly from Russell [DD79] and Poly. It is parametric dependent types. For example, a polymorphic sort routine could be defined by

```
let sort = proc[ t : type() with {less :proc(t,t ->bool)} ](A :*t)
        ! Your favourite sort routine using less
```

The first parameter *t* is a type which, since it is enclosed in square brackets may be implied from the other parameters at the time of call. The type *t* must have at least a function *less* defined in its signature which takes two objects of type *t* and returns a boolean. That is, we only require a partial match between signatures of the actual and formal type parameters. The second parameter *A* is a vector of objects of type *t* . To call the procedure we might use

```
sort( int with '<' as less, a vector of integers )
```

There is a little renaming here to allow the integer infix operation '<' to be used as *less* inside the procedure. Of course, we can partially apply the procedure to freeze the type. Thus

```
let int.sort = sort( int with '<' as less )
```

defines a function *int.sort* that may be applied to vectors of integers.

A more exciting prospect is the possibility of types themselves being parameterised by polymorphic values. For example, all pairs of values of the same type may by defined by

```
let all.pairs = type[ t :type() ]( first,second :t ) is first,second
```

Then all integer pairs could be defined by

```
let int.pairs = all.pairs( int )
```

All of the lists of elements of the same type can be defined by

```
let list = union[ t :type() ]type() |
           type( hd : t ; tl : list( t ) ) is hd,tl
```

Thus

```
let int.list = list( int )
```

defines the type of a list of integers. We can also define lists of mixed elements by not parameterising the second element of the list. For example

```
let list = union[ t : type() ] type() |
           type( hd : t ; tl : list( t ) ) is hd,tl

let nil = list( int )()
let intelement = list( int,3,nil )
let mixed.list = list( string,"ronald",intelement )
```

7. A DATA TYPE COMPLETE FORM OF RELATIONS

Relational constructs in a language are well known [SCHM78] [WSKRV81] [ROWE80] [SHOP79] and other persistent languages accommodate classes of entities [AOO82] [NAUR63] [SFL83]. The relational languages, however, typically restrict the types over which the relation may be formed, presumably to reduce implementation problems or to adhere to first normal form (1NF) [CODD70]. The notion of an atomic value, on which 1NF is based, is however relative; most such languages allow character strings as a column type, and also have a substring operation, for example. Consistent with our adherence with the principle of data type completeness, we allow a column to take any value permitted in the language.

7.1 Relation types

We intend the programmer to visualise the relation as a table, as a sparse array or as an extentionally defined partial function. For this reason, the type declaration is similar to that of a procedure i.e.

```
rel( parameters -> parameters )
```

The names and types in the first parameter list identify the columns that form the key of the relation. The names and types of the second parameter list identify the columns dependent on the key. Some examples of relation type declarations follow:

```
let rel1 = rel( i : int -> s : string )

let rel2 = rel( ai : *int -> x : #pixel ;
                y : proc( int -> rel2 ) )
```

A denotation for relational constants is achieved by using the type name, or type expression as a relational constructor. For example

```
let r1 = rel1{
              ! i  ->  s
              0    ->  "zero"
              1    ->  "one"
              2    ->  "two" }
```

7.2 Applicative relation operations

The usual relational algebra operations join, project, select, union, intersection and difference are defined over objects of this type. These are applicative, so they generate new relational instances, whose type can be deduced from the operands and operator.

7.3 Transfer operations

A difficulty facing the language designer for a language with bulk data structures is the design of an interface mechanism. We use indexing, an iterator and a de-setting operator.

Indexing is written exactly like function application or array subscripting. For example

```
r1( 2 )
```

would yield the 'result' part of the tuple with index value 2 in relation r1. In general a sequence of expressions provides values for an equality match on the index columns (those appearing before the arrow). If there is no match, an exception is raised. If there is a match then the result is an actual store, selected by this index, with the environment corresponding to the signature appearing after the arrow in the relation type definition defining the formal store for compilation. Thus it may be used exactly like an ADT, both in the LHS and RHS contexts. As the index environment does not appear in this store, the semantic and engineering problems of update to an index are avoided.

The iteration is written

```
for each x in y do statement
```

This is a general form applicable to arrays and relations. The variable *x* takes all the values in turn, in some arbitrary but repeatable order, corresponding to the components of *y* . In the case of a relation it takes the value of an ADT with environment formed by the concatenation of the index and result signatures. The statement is compiled in the context of a formal store defined by this environment, with the index fields protected as constant, to prevent update to the index. The *statement* is executed once with each actual store corresponding to a tuple in the relation.

De-setting is written

```
the expression
```

The expression has to have a relational type, and typically it is a selection expression e.g.

```
the r1 where s = "one"
```

If the value of the expression is not a singleton set then an exception is raised. If it is, then an actual store corresponding to the tuple, with index fields protected against update, is the result of the whole construct. It may be used in a LHS or RHS context.

The transfer interface in all three cases is based on a statically typed, but dynamically selected and bound store substitution. Thus it provides for individual access and update operations on the bulk data.

7.4 Bulk update operations

In situ update of bulk data is deemed necessary for semantic (data sharing) and engineering reasons. In situ update via the transfer operations above already exists. Expressive power and engineering advantage can be gained by providing bulk update statements. These are

update *x* **adding** *y*

update *x* **removing** *y*

update *x* **substituting** *y*

The effect of the first is to make *x* now have the union of the tuples in *x* and *y* . The effect of the second is to remove all tuples from *x* that have matching tuples in *y* . The third substitutes for columns in the result of *x* new values from *y* for the types whose indexes match in *x* and *y* .

7.5 Avoiding a semantic anomaly

There is a well known update anomaly, to do with updates altering selection criteria, typified by discussion about updates such as

"fire all managers who earn less than l.5 times
the average salary of their immediate subordinates"

We take the view that these semantic ambiguities are best removed by making the code equivalent to identifying all the updates with the data unchanged, and then performing the updates. It is supported in the language by a dynamic lock on relations, so that, while they are the subject of a selection or iteration scan, no in situ updates may be performed.

8 CONCLUSION

Name spaces have been proposed as a mechanism to formalise the association between stores and program. They are motivated by our observations on longer term storage. Static binding mechanisms are supported, which we see as sufficient for small systems and short term data. A dynamic mechanism for binding actual stores to formal stores is proposed to accommodate the changes inevitable in longer term systems which cannot be economically dealt with by total rebuilding in large systems.

The type relation has been presented as a bulk data representation, with the usual restrictions relaxed in favour of data type completeness. The abstract data type, union and parametric polymorphic types are presented to allow precise data description. The combination of these with the type and binding rules for name spaces is shown to be simple and consistent.

The model proposed enables the whole program to be statically bound and typed except for well identified points which may be introduced by the programmer if the flexibility is required. Even when the dynamic construct is used the type checking, though delayed, is no less strict.

9. REFERENCES.

[AOO82] Albano, A., Occiuto, M.E. & Orsini, R. A uniform management of temporary and persistent complex data in high level languages. DATABASE Infotech State of the Art Report 9, 8 (1982), 435-458.

[ABCCM83] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach to persistent programming. Computer Journal 26, 4 (1983), 360-365.

[AB85] Atkinson, M.P & Buneman, O.P. Database programming language design. in preparation.

[ABCCM81] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. Progress with persistent programming, **Database - role and structure**, Cambridge University Press, Cambridge, 1984.

[AM84] Atkinson, M.P. & Morrison, R. First class functions are enough. 4th Conference on the Foundations of Theoretical Computer Science and Software technology. Bangalore, India. (1984).In Lecture Notes in Computer Science, 181 (1984), 223-240. Springer-Verlag.

[PS85] PS-algol reference manual. Universities of Glasgow and St Andrews PPRR-12 (1984).

[BL84] Burstal, R. & Lampson, B. A kernal language for abstract data types and modules. Proc. international symposium on the semantics of data types, Sophia-Antipolis, France (1984).

[CARD85] Cardelli, L. Amber. A.T. & T. Bell Laboratories (1984).

[CODD70] Codd, E.F. A relational model for large shared databases. Comm.ACM 13, 6 (1970), 377-387.

[DD79] Demers, A. & Donahue, J. Revised report on Russell. Technical report TR79-389, (1979), Cornell University.

[ICHB83] Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).

[LAND66] Landin, P.J. The next 700 programming languages. Comm.ACM 9, 3 (1966), 157-164.

[LHJSW83] Liskov, B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R. & Weihl, W. Preliminary Argus manual. Technical Report Momo 39 (1983), M.I.T.

[MATT85a] Matthews, D.C.J. Poly manual. Technical Report 65 (1985), University of Cambridge, U.K.

[MCAR62] McCarthy, J. et al. Lisp 1.5 Programmers manual. M.I.T. Press Cambridge Mass. (1962).

[MILN84] Milner, R A proposal for standard Ml. Technical Report CSR-157-83 University of Edinburgh. (1983).

[MW80] Mylopoulos, J. & Wong, H.K.T. Some features of the Taxis data model. 6th International conference on Very Large Data Bases. (1980). Montreal.

[NAUR63] Naur, P. et al. Revised report on the algorithmic language Algol 60. Comm.ACM 6, 1 (1963), 1-17.

[ROWE80] Rowe, Reference manual for the programming language RIGEL. Technical report. University of California at Berkeley. (1980).

[SCHM77] Schmidt, J.W. Some high level language constructs for data of type relation. ACM.TODS 2, 3 (1977), 247-261.

[SCHM78] Schmidt, J.W. Type concepts for database definition. in **Databases:Improving usability and responsiveness**. Academic Press. (1978).

[SHOP79] Shopiro, J.E. THESEUS - a programming language for relational databases. ACM.TODS 4,4 (1979).

[SFL83] Smith, J.M., Fox, S. & Landers, T. Reference manual for ADAPLEX. Computer Corporation of America, Cambridge, Massachusetts (1981).

[TENN77] Tennent, R.D. Language design methods based on semantic principles. Acta Informatica 8 (1977), 97-112.

[TURN79] Turner, D.A. SASL language manual. University of St.Andrews CS/79/3 (1979).

[WSKRV81] Wasserman, A.I., Sheretz, D.D., Kersten,M.L., van de Reit, R.D. Revised report on the programming language PLAIN. ACM SIGPLAN Notices 16, 5 (1981), 59-80.

[WIRT71] Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

[VANW75] van Wijngaarden, A. et al. Report on the algorithmic language Algol 68. Numerische Mathematik 14,1 (1969), 79-218.