Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store

Malcolm P. Atkinson and Ronald Morrison

University of Glasgow, Glasgow, Scotland G12 8QQ. Tel. 0413398855

University of St Andrews, St Andrews, Scotland KY16 9SX. Tel. 033476161

1. Introduction

At the first Appin workshop we left unresolved the problem of how to write general utility programs over objects in the persistent store [2]. We said

> "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."

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

In totally dynamic systems, there is less of a difficulty in resolving this problem. When types and names are first class values it is relatively easy to construct abstractions over types and names that will allow such iterations. However, this is at some cost since we now have dynamic type checking and dynamic name resolution.

In this paper we will investigate a method of constructing iterations over persistent objects while still retaining static type checking and name resolution. In doing this we are essentially trying to discover where the magic is in a system and who is it safe to give the magic to. In a type secure object store, the security of the system depends upon the type checking being safe. To ensure this, it is usual for the type checking to be performed by one trusted module in the compiler. This module converts program text into typed objects (by magic) by using a lower level technology which it only knows about. In operating systems, capability addressing uses a similar mechanism[6]. The question is when and where is it safe to use this magic?

One technique that is used in the type secure PS-algol store[9] is to have available, as an object in the persistent store, the compiler itself. Thus to write a browser we can automatically construct a program according to the object we are trying to browse and submit that program to the compiler[5]. The compiler supplies the magic by turning the text description of the type of the object into the form used by the system. This technique is very successful and can always be used as a fall back position. However it is a little cumbersome when we wish to write customised iterators over objects.

We will propose a compromise between security and flexibility by introducing first class names in a controlled manner. First we will describe the Napier[8] universe of discourse from which the object store is built.

2. Napier Types

The Napier type system is loosely based on one suggested by Cardelli & Wegner[4]. All data objects in the system can be described by the following rules

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

2. #pixel is the type of an image 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 types $t_1,..t_n$, $t_1|...|t_n$ is the type of a union of the 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.

7. type parameterisation may be used in the type algebra.

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

9. for any identifier I and any type t, **abstype** I **with** t, is the type of an abstract data type.

10. any procedure type may be universally quantified.

11. type **any** is the infinite union of all types.

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

We will describe the more important aspects of this type system as we need. The essential element is that there is a high degree of abstraction. Thus, in the above, vectors and structures are regarded as store abstractions over all data types, procedures as abstractions over expressions and statements, abstract data types as abstractions over declarations and polymorphism as an abstraction over type. The infinite unions **env** and **any** are used to support persistence as well as being a general modeling technique. The types picture, pixel and #pixel are used for graphics but have little relevance to our discussion here. The underlying philosophy of the Napier type system is that types are sets of values[4], and therefore type equivalence is decided by structure.

3. Persistence

We have defined the persistence of data to be the length of time for which data exists and is usable. Thus it is an abstraction over one of the physical properties of data, that of the length of time for which we keep it. Elsewhere[1,3,7] we have described the advantages of not having to explicitly program for the differences in the use of long and short term data and we will not labour them here. It is sufficient to say that by ensuring the persistence abstraction we obtain significant gains in the software engineering of large systems. The figure often quoted is 30% of the total cost of a system throughout its life cycle[1].

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

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

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

```
let e = environment ()    !create a new empty environment
let id =     proc [t : type] (x : t -> t) ; x
             in e  !Add the id to the environment e
let id_env = e in PS
       ! create a new environment in PS.
       !This new one is the binding id_env :             ! env
```

Figure 1: Binding to an environment

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

To retrieve 'id' for reuse we could write

```
use PS as id_env : env in
       use id_env as id : proc[t : type] (t  -> t) in
       begin
              let Y = id [string] ("Stewart Hotel")
              write Y
       end
```

Figure 2: Reusing objects in an environment

The **use** clause binds an environment and some of its field names to the clause following **in**. For example

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

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

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

The Napier persistent store comprises of objects that are statically bound. However these statically bound objects may be dynamically composed into larger objects. We would expect most small components to be statically bound for safety and larger units of system construction to be dynamically composed. The choice is with the programmer. However, our skill in efficient and effective system construction will be in deciding which components are complete and may be statically bound and which components are better dynamically composed.

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

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

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

4. First Class Names

We propose to augment the Napier type system with a new type called name. A name is an identifier with a fixed type. As we will see later the fixed type is necessary to preserve the type security of the object store. We will introduce the concept by example.

```
type Ron is structure (field1, field2 : int ; field3 : string)
let Malcolm = Ron (1, 2, "Appin")
```

This introduces a structure type called 'Ron' and an instance of it called 'Malcolm'. We can index the object by, for example

```
Malcolm (field3)        which would yield "Appin"
```

We can also introduce a variable name object by

```
let Name1 := string_to_name [int] ("field1")
```

'Name1' is of type name [int]. That is, it is a name that can only be used in the context of an integer index. Thus

Malcolm (Name1)   yields 1

Furthermore we can assign other objects of type name [int] to the variable 'Name1'. For example

Name1 := string_to_name [int] ("field2")

and thus

Malcolm (Name1)   now yields 2.

The operations on names consist of converting stings to names and visa versa by the following built in procedures.

```
let string_to_name = proc [t : type] (s : string -> name [t])
! convert a string to a name of type t

let name_to_string = proc [t : type] (n : name [t] -> string)
! convert a name of type t to a string
```

Two names are equal if they have the same type and denote the same name. Thus

```
let one = string_to_name [int] ("field2")
let two = string_to_name [int] ("field2")
one = two              yields true
```

Two names are assignment compatible if they have the same type and finally the type of a name is written as

name [<type>]

It should be noticed that we have let very little of the magic out by this mechanism. The compiler's ability to convert strings to names has been provided as a general built in procedure. We have not allowed strings and names to be freely mixed, and therefore confused, but have insisted on transfer functions to convert one to the other.

With the above facilities we can interactively read in a field name and use it to index a structure. For example

```
  let index_field = proc (This : Ron -> int)
  ! read in an integer name and index the object 'This' by it returning the     !
integer value obtained
  begin
        write "What is the field name?"
        let n = string_to_name [int] (read_string ())
        This (n)
  end
```

Figure 3: Index a structure by a read in field name

If the field name is not available in the object then an exception is raised.

5. Iterators

Our next extension to Napier is to provide a built in iterator over environments. An environment is a collection of bindings consisting of a type, a constancy attribute, a name and a value, all of which we would like to abstract over. We will investigate iteration by example.

First we will write a procedure to copy one field from one environment to another. As in all the following examples we will ignore the exceptional conditions when the desired field is not available in the indexed environment or we have a duplicate field. Although the programming for this is simple it constitutes noise in the examples.

```
let copy_one_entry = proc [t : type] (env1, env2 : env ; n : name [t])
! copy the field with name n from env1 to env2
      if constant (env1 (n))   then    add n = env1 (n) in env2
                               else    add n := env1 (n) in env2
```

Figure 4: Copy one field in an environment

In this example we have introduced some new ideas. First we have passed a name as a parameter. The predicate 'constant' allows us to test whether the binding is constant or variable so that we can copy it correctly. The reserved word **add** allows us to add a new binding to an environment. **add** operates on names whereas **let** operates on identifiers.

We could copy only the constant fields of one environment into another but for this we require an iterator.

```
let copy_constant_fields = proc (env1 : env -> env)
begin
    let env2 = environment ()
    for each  t : type, n : name [t] -> v in env1 do
        if constant (env1 (n)) do add n = v in env2
end
```

Figure 5: Copy constant fields

The iterator provides us with an identifier for the type, the name and the value for each of the fields in the environment. The name may be used as a first class name but the type may not, as types are not values in the system. However, the type is still useful as can be seen from the next example to merge two environments.

```
let merge_environments = proc (env1, env2 : env)
! add to env1 all the non duplicate bindings of env2
begin
    let duplicates := false
    for each t : type, n : name [t] in env2 do
    begin
        if n in env1    then duplicates := true
                        else copy_one_entry  [t] (env2, env1, n)
        if duplicates do raise name_clashes ()
    end
end
```

Figure 6: Merge two environments

We have introduced the infix operator **in** here to test if a field is in an environment. Notice also that in the loop we do not specify the value. The type 't' is used in calling 'copy_one_entry'. Notice that as the value is not used we have omitted it from the iterator. The name and type can be similarly omitted.

The next example iterates over an environment displaying only the integer field names.

```
let display_int_field_names = proc (env1 : env)
    for each n : name [int] in env1 do
        write name_to_string [int] (n)
```

Figure 7: Display integer fields

Thus we can iterate selectively over the field according to the type of the name. In our next example we read in the name and the value of an integer field and add it to an environment.

```
let add_int_field = proc (env1 : env)
! read in an integer field and add it to the environment
begin
    write "Is the field constant?"
    let constancy = read_reply ()
    write "What is the field name?"
    let n = string_to_name [t] (read_string ())
    write "What is the value?"
    if constancy    then add n = read_int () in env1
                    else add n := read_int () in env1
end
```

Figure 8: Add an integer field to an environment

Finally we will write a procedure to display all the information in an environment.

```
let display_environment = proc (env1 : env)
    for each t : type, n : name [t] -> v in env1 do
        write name_to_string [t] (n), ":", typeof t, "=", v, "'n"
```

Figure 9: Display the information in an environment

This procedure can be adapted to selectively iterate over all the fields. The reserved word **typeof** is only applicable to type in the **for each** iterator. This allows us to discover a string representation for the type in a safe manner. Thus we can perform selective actions depending upon the type without resorting to types as values and dynamic type checking.

6. Conclusions

We have introduced the notion of a first class name by example showing how they can be used to index structures and environments in Napier in a manner consistent with the type system of the language. We have also introduced a polymorphic iterator over environments to allow us to write the general utility programs normally associated with file stores. We feel that in a polymorphic language it is entirely consistent to have polymorphic iterators and intend to explore their utility further with regard to other operations and data types.

For the present we feel we have reached a reasonable compromise in providing file store facilities and retaining the security of the type system across the persistent object store. We stopped short of having types as values to avoid more dynamic type checking. We can always provide these facilities by calling the compiler.

Finally where is the magic? We have clearly left the magic with the compiler and the two name conversion procedures. Thus the system is secure if these modules are.

7. Acknowledgements

We acknowledge the many discussions with our collaborators on the PISA project. In particular, Al Dearle for his work on the design and implementation of environments. Dolphins are sometimes understood. The work is supported by SERC grants GR/D 4326.6 and GR/D 4325.9 and a grant from STC.

6. References

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

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

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

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

5. Dearle, A. & Brown, A.L. Safe browsing in a strongly typed persistent environment. Universities of Glasgow and St Andrews PPRR-33, (April 1987).

6. Dennis, B. & Van Horn, E.C. Programming semantics for multiprogrammed computations. Comm.ACM 9, 3 (March 1966), 143-155.

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

8. Napier language reference manual. in preparation

9. PS-algol reference manual. 4th Edition. Universities of Glasgow and St Andrews PPRR-12 (July 1987).