The Octopus Model and its Implementation

Alex Farkas and Alan Dearle

{alex,al}@cs.stir.ac.uk
Department of Computing Science
University of Stirling
Scotland, FK9 4LA

Abstract

The paper describes a new reflective language mechanism and its implementation. The mechanism called Octopus is designed to support a wide variety of database programming and software engineering applications which previously required much heavier weight mechanisms, such as the use of a compiler at run-time, or unsafe language mechanisms. The implementation of this mechanism requires structured type representations to be available for manipulation at run-time and architecture support for *boxed* values. These implementation techniques are described and the cost of these mechanisms is examined.

1 Introduction

In most programming languages, programs and data form directed graphs with nodes consisting of arbitrary values, such as program fragments (e.g. procedures), records and arrays. Scalars form the leaf nodes of the graphs; they may be referenced but do not themselves reference other values. Figure 1.1 shows a conceptual view of such a graph in which the nodes represent values and the arcs represent bindings between values. In general, bindings have four components: a name, a value, a type, and an indication as to whether or not the value may be modified [10].

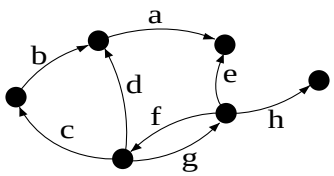

Figure 1.1: A binding graph.

Applications consist of programs which construct or navigate these graphs. Usually, the types of the nodes over which programs operate are statically known by the application programmer. Another class exists in which the programmer does not know the types of the data which may be encountered by the program.

For example, the types of values that applications such as object browsers and query tools may encounter are unpredictable. Clearly some dynamic typing is required in these cases. However, few programming languages provide mechanisms through which the types of these values can be enumerated or discovered, if they are unknown.

One solution to these problems is to make use of some form of *linguistic reflection* [13]. Reflective systems permit their own structures to be examined and altered from within. Linguistic reflection has two basic forms: compile-time and dynamic. Both permit the construction of new program elements from within another program, the difference is when reflection is performed.

Compile-time linguistic reflection is akin to having a macro processor for the host language which displays an understanding of the semantics of that language. When dynamic linguistic reflection is employed, new programs are constructed dynamically and introduced to a running program. For example, when the PS-algol [4] and Napier88 [9] browsers encounter a value of a type which has not previously been encountered, they construct, compile and execute a new program (procedure) to display the value.

2 The Octopus Model

Octopus is a new dynamic linguistically reflective mechanism which provides a dynamic infinite union type with a set of reflective operations. These operations may be used to manipulate values of any type without the expense of the dynamic techniques described above. Octopus provides a uniform abstract interface to values of any type, this facilitates a number of higher level activities, namely:

- construction of browsing tools,
- software debugging,
- querying over complex objects,
- evolution of programs and data, and
- distribution of complex object closures.

Octopus is an acronym for Object Closure Transplantable to Other Persistent User Spaces. As the name suggests, the technique also provides the ability to isolate portions of closures, and copy them to other locations, thus facilitating software component and data distribution. Partial closures may be *cut* from one location and *rewired* in another, possibly in a different context, using the interface supplied by Octopus.

A brief description of Octopus is given in this section, and the reader is referred to [5] for a more detailed discussion on the use of this mechanism. The essence of the Octopus mechanism is to allow values from the programming language value space to be *hoisted* up to a meta level and manipulated in ways which the programming language would not otherwise permit. This is achieved using the *coerceToOctopus* operation. When manipulation is complete, an Octopus may be *dropped* back into the value space, provided the encapsulated values still conform to the language's type system. Dropping is performed using the *coerceFromOctopus* procedure†. Type signatures of these two operations are shown below.

```
coerceToOctopus   : proc( Value → Octopus )
coerceFromOctopus : proc( Octopus → Value )
```

In the procedure signatures above, the value hoisted to and dropped from an Octopus has the type *Value*. Since values of any type may be represented as an Octopus, the type *Value* must be an infinite union type, and type checking must be performed dynamically. In Napier88 [11], this functionality is delivered by the infinite union type *any*, into which values of any type may be injected. Values encapsulated in an *any* are type compatible with each other even if the encapsulated values are of different types.

In the Octopus model all values conceptually have special mappings, or *wiring diagrams*, associated with them which contain information about the bindings within those values. Wiring diagrams are normally inaccessible to programmers; however, they may be made available using the hoisting procedure. An Octopus may be thought of as a uniform viewing mechanism with which values of any type and their associated wiring diagrams may be viewed and manipulated. An Octopus is implemented as a package of functions contained in a structure; the hoisted value is encapsulated in the closure of these functions and the functions operate on the value's wiring diagram. The type declaration for an Octopus is shown in Figure 2.1

```
type Octopus is
    structure( getType:   proc( → TypeRep );
               getSource: proc( → Source );
               scan:      proc( proc( Binding ) )
               )
```

Figure 2.1: The structure of an Octopus.

The *getType* operation returns a representation of the type of the value encapsulated in the Octopus. This representation is a value in the programming language space and may not be used as a denotation for a type. The nature of these representations is described in more detail in Section 3.

The *getSource* operation returns a representation of the source code for the value. If the value is a procedure, this source code is similar to the hyper-program model of source code described in [6], [7] and [8]. If the value encapsulated in an Octopus is not a procedure, then *getSource* returns a representation of the value which is suitable for use in hyper-programs.

A *scan* procedure is provided to iterate over the bindings contained in an Octopus; *scan* takes as its single parameter a programmer specified procedure which is iteratively applied to each binding in the Octopus. The specified procedure may perform an arbitrary computation on a binding; for example, it may display a binding's name or type.

```
type Binding is
    structure(
        cut      : proc( → bool );
        add      : proc( Value → bool );
        get      : proc( → Value );
        resolved : proc( → bool );
        getType  : proc( → TypeRep );
        getName  : proc( → string ) )
```

Figure 2.2: The representation of a binding.

Each binding is represented as a package of six operations, as shown above in Figure 2.2, which behave as follows:

cut — When applied, *cut* causes the associated binding to be dissolved. The process of cutting a binding is simply a meta level indication that the binding is no longer resolved. Cut bindings may still be accessed via direct bindings to the naked value.

add — The *add* operation allows an unresolved binding to be rewired, or resolved, using the given value. The operation fails if the binding is already resolved or if the supplied value is of the wrong type.

get — When applied, *get* returns the current value of the binding. If the binding is unresolved, a fail value is returned.

resolved — This operation returns *true* if the binding is in a resolved state and *false* otherwise.

getType — This operation returns a representation of the type of the corresponding bound value.

getName — This returns the name of the bound value.

† In this paper the word procedure is used synonymously with the word function.

```
type Person is structure(   name :   string;
                            age  :   int )
let aPerson = Person( "john", 42 )

let scanner = proc( b : Binding )
begin
  writeString( b( getName )() )
  if EqualType( INT, b( getType )() ) then
      writeString( " : int'n" )
  else
  if EqualType( STRING, b( getType )() ) then
      writeString( " : string'n" )
  else
      writeString( " : unknown type'n" )
end

let olly = coerceToOctopus( aPerson )
olly( scan )( scanner )
```

Figure 2.3: A program to browse values.

```
name : string
age : int
```

Figure 2.4: Output of the above program.

To illustrate the use of Octopus, consider the program in Figure 2.3 which displays the types of the fields of the record denoted by *aPerson*. The output of this program is shown in Figure 2.4. The procedure *scanner* displays the name and type of a binding; this procedure is iteratively applied to each binding in the Octopus *olly* using the *scan* operation. *scanner* obtains the type of each bound value using the *getType* operation of the binding and checks for type representation equality using the *EqualType* operation provided by the Napier88 system. In this example, *scanner* only has knowledge of two types, integer and string. However a more sophisticated version of this procedure could be written to display the type of an arbitrary value.

The following sections describe the implementation of the Octopus mechanism in Napier88. Section 3 describes the implementation platform provided by the Napier88 system, Section 4 describes an alternative implementation architecture more suited to the Octopus model. Section 5 describes how this architecture is used to implement Octopus, Section 6 gives some performance measurements and Section 7 offers some conclusions.

3 Napier88 Implementation

The Octopus mechanism has been prototyped using the Napier88 implementation platform. Therefore prior to describing the implementation of the Octopus mechanism per se, the Napier88 implementation platform is described. This section deals with two implementation issues pertinent to the implementation of the Octopus model, namely, the representation of types and the storage architecture of the system.

3.1 Type Representations

The Napier88 system provides a module devoted to manipulation of type representations. This module, known as the types module, provides a complete set of selector, constructor, equivalence and iterator functions that operate on type representations [2]. In practice, the types module implements an abstract data type which hides the representation of types. The Napier88 compiler makes use of this module to construct, manipulate and type check type representations during compilation. The constructor functions are primarily used at compile time to construct type representation such as the one shown in Figure 3.1.1. The selector functions are primarily used by the compiler to perform tasks such as the discovery of the types of fields of records and arrays. One strict equivalence predicate *EqualType* is provided, and is used at both compile time and dynamically. Another predicate *IsType* is provided that allows the class of a type to be discovered, for example whether the type represents a record, a procedure or an array.

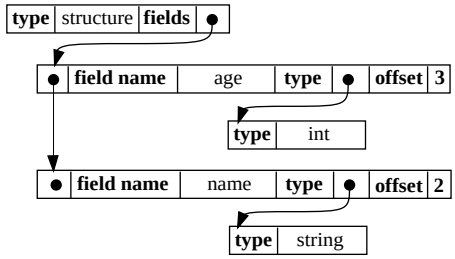

Figure 3.1.1: A type representation.

Many compilation systems use scalar values, such as integers, to represent compile time types. Rather than using this approach, the Napier88 types modules constructs graph representation of Napier88 types. This technique is more expensive than using scalars, however it is necessary due to a number of features of the Napier88 system.

Firstly, Napier88 makes use of structural type equivalence; two types which are syntactically different may represent the same type. Consequently, some structured canonical representation of the types must be compared in order to check equivalence. Secondly, the orthogonal persistence provided by Napier88 allows programs and data to be independently prepared and combined at a later time. Since there is no universal global mapping which maps types onto scalar representations, the type representations must contain (at least) all the information contained in the original specification of the types.

For example, consider the following type declaration shown in Figure 3.1.1 which declares a record type with two fields with labels *name* and *age* and types with types *string* and *integer* respectively.

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

Figure 3.1.2: A simple type declaration.

When this type declaration is compiled, a representation of the type is constructed, as shown in Figure 3.1.1 above. The type representation consists of a node indicating that the type is a structure. This node points to a list of nodes, each representing a field of the record type: the first node in the list indicates that there is a field called *age* which is an integer, and the second node indicates that there is a field called *name* which is a string. Field offsets are also recorded as part of the type representation; these are an extension to the types module described in [1] and the meaning of these is described later. The nodes are stored in a canonical (alphabetical) sequence in order to make type checking more efficient.

Type representations are primarily used by the compiler at compile-time. However, type representations are also required at run-time to support dynamic type checking. For example, the infinite union type *any*, described earlier, may encapsulate values of an arbitrary type. In the implementation of *any*, a representation of the type is stored alongside the encapsulated value. Such values may be dynamically projected onto their original types, this requires both the types module and the type representations are available at run-time.

3.2 The Persistent Abstract Machine

This section describes the Napier88 block retention mechanism and highlights those aspects which prevent the Octopus mechanism as specified in the introduction from being implemented.

Napier88 programs are compiled to produce code for the persistent abstract machine, PAM, which provides an interface to a persistent heap based storage architecture in which all data is stored in objects. Napier88 values are represented by combinations of scalar words and pointers to other heap objects. In order to make garbage collection and persistent object management more efficient, PAM objects conform to a canonical format with all pointers stored before scalars as follows:

word 0	object header + number of pointers
word 1	object size
words 2..n	pointer fields
words n+1..	non-pointer fields

Napier88 is a block structured language which supports first class procedures, that is procedures form part of the value space of the language. PAM creates a stack frame, stored in a separate heap object, for each dynamic block invocation. Each stack frame contains a *dynamic link*, pointing to the object containing the frame of the calling block, and a pointer to the object containing the frame of the lexically enclosing block, the *static link*. Values which are declared in statically enclosing blocks are accessed by following static links. The set of all frames reachable by following static links is known as the *static chain*.

In order to comply with the canonical object format, each frame contains two storage areas: one for pointers and one for scalars. These areas are used to store values created during the execution of the block and as expression stacks, they are known as the *pointer stack* and *main stack* respectively. The format of a PAM frame is shown in Figure 3.2.1.

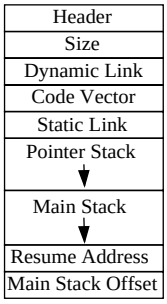

Figure 3.2.1: A PAM frame object.

When Napier88 values are declared, their pointer components are stored on the pointer stack and their scalar components are stored on the main stack of the currently active frame. For example, the Napier88 record data type, *structure*, is implemented as a pointer to a heap object which contains the fields of the record, with pointers stored before non-pointers.

Of particular importance with respect to Octopus is the implementation of procedures and the values encapsulated with them. Napier88 procedure values are stored on the pointer stack and are represented by two pointers collectively known as the *procedure closure*. The first of these is a reference to a code vector (a heap-object containing the executable code of a procedure). The second pointer is a reference to the frame of the statically enclosing block of the procedure. This pointer becomes the static link of the procedure when it is invoked. Procedure closures are formed dynamically by loading a pointer to the code vector of the procedure and a pointer to the current frame onto the current pointer stack.

3.3 Accessing a Static Environment

To illustrate the way in which a static environment is accessed, consider the following program, which declares a string, and two procedures, *warning* and *error*, which make use of that string.

```
let prefix := "**** "

!** A procedure for displaying warning messages.
let warning = proc( s : string )
      writeString( prefix ++ s )

!** A procedure for displaying error messages.
let error = proc( s : string )
      writeString( prefix ++ s )
```

Both procedures, *warning* and *error*, are bound to the string *prefix*. A conceptual view of this is shown in Figure 3.3.1, in which the arcs denote the bindings between the respective values.

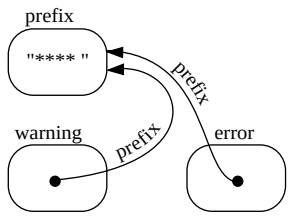

Figure 3.3.1: *display*, *message* and *error*.

In the implementation of Napier88, procedure closures are stored in the frame of the block in which they are declared. Therefore the string denoted by *prefix* and the closures associated with *warning* and *error* will be stored in the same heap object as shown in Figure 3.3.2. In order to use *prefix*, *warning* and *error* must follow their static links to obtain the value.

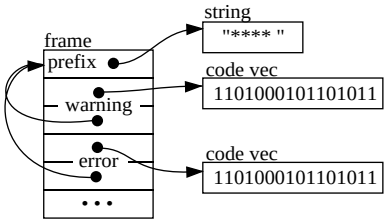

Figure 3.3.2: *prefix*, *warning* and *error*.

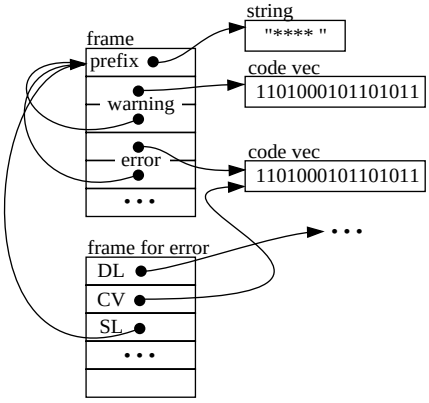

Figure 3.3.3 A call to procedure *error*.

This representation allows values declared in statically enclosing blocks to be shared. Figure 3.3.3 shows an invocation of the procedure *error*; the frame corresponding to the call of *error* can access the values of *prefix*, *warning* and *error* itself. It is precisely this sharing that prevents this mechanism from being used when Octopus is employed. Octopus permits bound values to be cut and replaced with different bound value. For example, the binding from *error* to *prefix* could be replaced by another string as shown in Figure 3.3.4.

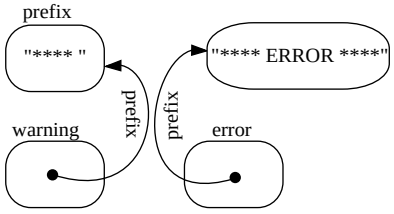

Figure 3.3.4: The updated procedures.

```
!** Procedure to wire new binding.
let cutAndRewire = proc( b : Binding )
  if  b( getName )() = "prefix" and
      EqualType( b( getType )(),STRING ) do
  begin
      let ok = b( cut )()
      if ~ok do error( .. )
      ok := b( add )( "**** ERROR **** " )
  end

let octopus = coerceToOctopus( error )
octopus( scan )( cutAndRewire )
```

Figure 3.3.5: A program to rewire a new *prefix*.

In Octopus, this is achieved using a program such as the one in Figure 3.3.5. In this program a procedure *cutAndRewire* is declared which examines a binding to see if it has type string and has the name *prefix*. If so, the binding is cut and replaced with the new value, "**** ERROR **** ". Finally, the *error* procedure is hoisted into an Octopus and the *cutAndRewire* procedure is supplied to the *scan* operation of the Octopus.

Using the Napier88 implementation described above, the desired semantics could not be achieved: it is not possible to alter the value bound to the procedure *error* without also changing the value bound to *warning*. In order to implement Octopus as described above, the implementation architecture of the Napier88 system requires modification.

4 Block Retention in PCASE

The main drawback of PAM as a vehicle for supporting the Octopus model is that parts of a static chain may be shared by any number of different procedures. Manipulating a single frame in a static chain may affect the operation of any procedure which shares that chain.

One solution to this problem is to avoid the use of a static chain altogether. This method is employed in the PCASE (Persistent Code vector, Argument, Stack and Environment) model [3] in

which the static chain is replaced by a *flat environment*, a single record containing pointers to each value used within a procedure. The construction of this environment is performed at procedure closure formation time. Using this technique, each value must be *boxed* [12]; that is encapsulated in a heap object. Closures consist of a pointer to a code vector and pointer to a record, the *environment vector*, containing pointers to the boxed values.

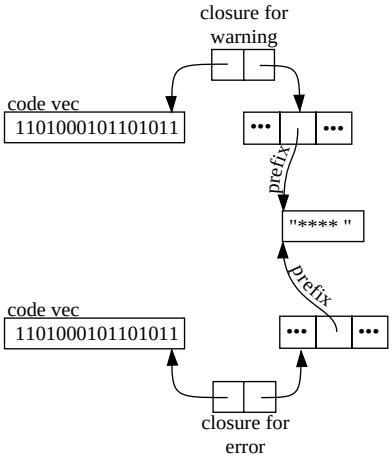

Figure 4.1: The *warning* and *error* procedures in PCASE.

In the PCASE model, the *warning* and *error* procedures described earlier are represented as shown in Figure 4.1. Clearly the environments for *warning* and *error* are divorced, allowing bound values in the static environment of one procedure to be manipulated without affecting the semantics of the other. For example, each binding labelled *prefix* in Figure 4.1 may be made to refer to a different boxed value. A variation approach is used in the Octopus implementation and is described in the next section.

5 An Architecture to Support Octopus

5.1 Block Retention

In the PCASE model, a flat environment is created for each anonymous block and procedure closure in a program. Within a single procedure invocation, an arbitrary number of anonymous blocks may be entered; using the PCASE model this would require a flat environment to be constructed for each anonymous block entered. This expense is not required to support Octopus since anonymous blocks cannot be treated as values.

The prototype implementation of Octopus described in this paper is therefore implemented upon a hybrid architecture which uses PCASE style flat environments within procedure closures and retains static links for anonymous blocks. A procedure closure consists of a pair of pointers; one to an environment vector and another to the corresponding code vector.

5.2 Wiring Diagrams

To allow bindings to be manipulated, the higher level operations provided by Octopus require the presence of wiring diagrams describing those bindings. The nature of a wiring diagram depends on the type of the hoisted value, in particular whether the values are procedural or non-procedural. For a non-procedural value, information about that value, and hence a wiring diagram, may be deduced from its type. For procedural values, wiring diagrams must be constructed at compile-time and stored in their code vectors.

As described earlier, the Napier88 types module may be used to discover information about a type representation and Octopus utilises this ability. Values that are injected into an Octopus using the *coerceToOctopus* procedure are always encapsulated within a dynamic type. These dynamic types always carry a representation of their type with them. When a value is hoisted, the type representation is extracted from the dynamic type and examined using the *IsType* predicate from the types module to determine if the value is procedural or non-procedural. The implementation of the Octopus operations is determined by the class of the extracted type.

5.3 Non-procedural Values

Type representation contain complete information for non-procedural types, for example, it is possible to determine the names, types and number of fields in a record. It is also possible to discover the field offsets in the objects representing instances of these types. This information is presented in a sanitised manner via the operations on bindings in an Octopus as shown in Figure 2.1.

However, wiring diagrams contain more information than that stored in type representations. In particular, the state of bindings within a value (i.e. whether they are cut or not) must be recorded. For example, in the case of a Napier88 structure, an array of boolean values is maintained indicating the state of each field. A pointer to this vector is stored in a field of the structure which is invisible to the application programmer, and only accessible via the Octopus operations. To illustrate this, consider the type *Person* declared in Figure 3.1.1. Suppose that an instance of this type is created as shown below.

```
let aPerson = Person( "fred",87 )
```

The resulting value is represented by the graph of objects shown in Figure 5.3.1.

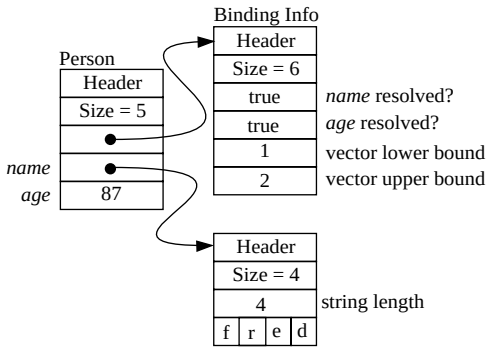

Figure 5.3.1: Representation of *aPerson*.

5.4 Wiring Diagrams for Procedures

Unlike non-procedural values, the representation of the type of a procedure does not contain sufficient information to enable the bindings within the procedure to be determined. Therefore, to obtain the wiring diagram for a procedure, it is necessary for the compiler to record information about the bindings within that procedure. This information needs to be available at run-time; this may be achieved by the compiler planting wiring diagrams in the code vector of every procedure.

Wiring diagrams may be thought of as a table containing one entry for each binding in the procedure. Each entry contains the name of the binding, its type and its offset within the procedure's environment vector. Each entry is created by the compiler upon encountering a reference to a value which is declared at an earlier lexical level than the procedure, i.e. if the value is declared outside the procedure body.

name	type	offset	state
prefix	string	2	resolved
writeString	proc(string)	3	resolved

Figure 5.4.1: The wiring diagram for *warning*.

For example, consider the program shown in Figure 3.3.1. The procedures *warning* and *error* are both bound to two values: *writeString* and *prefix*. The wiring diagram created for the procedure *warning* is as shown in Figure 5.4.1.

Like the information recorded for non-procedural values this information is used by the higher level Octopus operations.

5.5 Implementation Overview

To illustrate the combination of the various features described above, consider the following program in which the *warning* procedure declared earlier is hoisted into an Octopus, *octopus*, and the binding to the string *prefix* cut.

```
let octopus = coerceToOctopus( warning )
let scanner = proc( b : Binding )
begin
  if b( getName )() = "prefix" do
  begin
      let ok = b( cut )()
  end
end

octopus( scan )( scanner )
```

The result of executing this program is shown in Figure 5.5.1, in which the wiring diagram for *warning* contains a *dissolved* state for the binding named *prefix*.

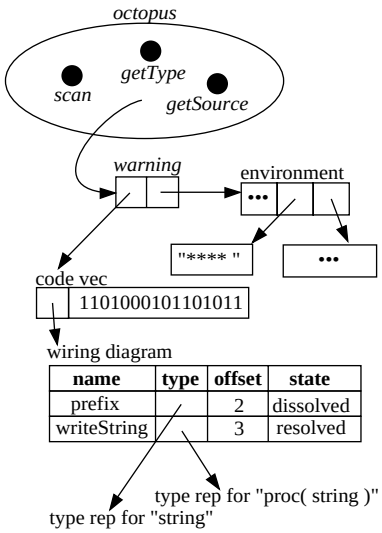

Figure 5.5.1: An Octopus representing *warning*.

The pointer from the environment vector of warning to the string denoted by prefix remains intact, even though the state of the binding is considered cut at the meta level. Therefore, any value bound to warning may safely execute the procedure without adverse effects. The difference is that if *octopus* is transplanted, then the string associated with the name *prefix* is not copied and must be rewired before the encapsulated procedure may be extracted. When a new string is wired in then all values bound to *warning* will instantly see the new value.

6 Performance

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

type System is
     structure( add:   proc( string,int );
                find:  proc( string → Person )
)

let nilPerson = Person( "",-1 )
let database := vector 1 to SIZE of nilPerson
let current := 1

let addPerson = proc( name : string ; age : int )
begin
     database( current ) := Person( name,age )
     current := current + 1
end

let findPerson = proc( name : string → Person )
begin
     let tmp := 1
     let found := false
     while tmp <= current and ~found do
     begin
         if database( tmp )( name ) = name then
             found := true
         else
             tmp := tmp + 1
     end
     if found then database( tmp )
     else nilPerson
end

in PS() let anExampleDataBase =
     System( addPerson,findPerson )
```

Figure 6.1: A simple Napier88 application.

The use of the hybrid block retention architecture described above has an effect on the performance of the Napier88 system. Some of these effects enhance performance, some are detrimental.

The most notable performance cost of the hybrid architecture is the requirement to wrap each declared value in a heap object. For example, integers which are directly stored on the main stack of the current frame in the PAM architecture, are wrapped in a heap object in the hybrid. This incurs a two fold cost – firstly an additional heap object must be created, secondly an additional dereference is required on use. However, the hybrid technique is more efficient for looking up intermediate free variables since they can be found using a single dereference of the environment vector rather than a search of the static chain.

In the hybrid architecture, the formation of procedure closures is considerably more expensive than the PAM architecture. At the point of closure formation, each value referenced by the procedure must be loaded into a newly created environment vector. This expense is not as bad as it might first seem, since it is only incurred when the closure is formed and this is a relatively infrequent occurrence. Calls to the procedure are no more expensive than using the PAM.

One benefit of the use of a flat static environment is the reduction in the amount of retained data in procedure closures. One of the drawbacks of the PAM architecture is that all values in the static chain are retained whenever a procedure closure is created. Since Napier88 programs commonly use higher order procedures to provide interfaces to encapsulated data this is a serious consideration. This problem is removed by only retaining that data which is used by a procedure.

Consider the simple Napier88 database application program in Figure 6.1 which maintains an database of records of type *Person*. This program is encapsulated in a package of type *System* containing two procedures: *add* and *find* which, respectively, create a new person and find an existing person. This package is placed in the persistent store by the last line of the program. The database is implemented as a vector of records and is restricted to no more than *SIZE* entries. The procedures *addPerson* and *findPerson* operate over this vector. These procedures maintain a reference to the vector and an integer representing the start of free space in their closures.

The following sections attempt to quantify the performance of the hybrid architecture with that of the original PAM architecture. Performance is compared with respect to three metrics: the amount of retained data, space utilisation and speed.

6.1 Block retention

The first measurement illustrates the amount of retained data in the two implementations. These measurements are taken with the size of the vector being ten (*SIZE*=10) and no entries in the database. Using PAM the total number of objects within the closure of the database is 29. These objects are distributed as shown in the histogram in Figure 6.1.1. The large peak of objects of size 6 and 9 reflect the retention of the type representations for *Person* and *System*. The object of size 169 is the code vector for the main program. These objects do not strictly need to be retained in order for the application to execute.

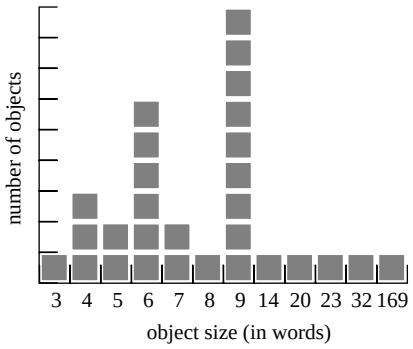

Figure 6.1.1: Object distribution in PAM.

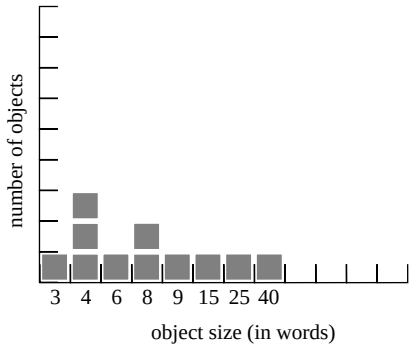

Figure 6.1.2: Object distribution in hybrid.

The hybrid architecture produced the results indicated in Figure 6.1.2 which has a dramatically different object size distribution. Note that there are only 11 objects retained rather than the 29 for the PAM implementation. This reflects the fact that this architecture retains only that data actually needed by the application. The object size distribution is also dramatically different; the two largest objects represent code vectors, the object of size 15 is the vector containing the database. The smaller objects represent a record, boxes and environment vectors.

6.2 Space Utilisation

Size	Number
3	1
4	4003
5	4002
6	6
7	2
8	1
9	9
20	1
23	1
32	1
182	1
4004	1

Total number of objects = 8029
Total size of objects = 40425 words

Figure 6.2.1: Space Utilisation in PAM.

The next performance test measures the overall space efficiency of the two systems. In this test, the size (*SIZE*) of the vector was set to 4000 and the database was populated with random data. Figure 6.2.1 shows the object size distribution for the PAM implementation whereas Figure 6.2.2 represents the distribution for the hybrid. The total space used is roughly equal in both systems; the dominant factors in this application are the records in the vector (and the strings contained in them) and the vector itself. These are represented by the 4000 objects of size 4–6 in the tables and the single object of size 4004(5).

Size	Number
3	1
4	4003
6	4001
8	2
9	1
25	1
40	1
4005	1

Total number of objects = 8011
Total size of objects = 44100 words

Figure 6.2.2: Space Utilisation in hybrid.

6.3 Speed

The speed of program execution was measured by testing the time taken to load 4000 items into the database. With the PAM system this took an average of 0.4417 seconds, with the hybrid system this took 0.6583 seconds. Thus for insertion the hybrid system suffers a 50% increase in execution time. When an instruction trace is taken, it is revealed that this time consists almost entirely of calls to the object allocator. This is entirely due to the boxing of values in the hybrid.

A second measurement was made by testing the time taken to lookup an item in the database 10000 times. The results were consistent with those above in that the hybrid system displayed an approximately 50% increase in execution time. These results reflect the intrinsic cost of object allocation in the Persistent Abstract Machine architecture; the additional 50% increase in execution time stems largely from the layered, object-based architecture of the PAM. An investigation of an alternative PAM architecture [14] is the topic of further research.

7 Conclusions

This paper describes the Octopus mechanism and the issues relating to its implementation. The Octopus mechanism provides a uniform abstract interface to values of any type. The approach is to provide a simple, yet powerful reflective mechanism and to use this to construct higher level

tools. This relatively simple mechanism provides enough power to allow many reflective applications to be written which previously required much heavier weight mechanisms, such as the use of a compiler at run-time, or unsafe language mechanisms. In particular, the Octopus mechanism may be used to support database operations such as browsing and querying. It may also be used to support a variety of software engineering tasks, such as debugging, incremental system evolution and distribution of software components. These are currently a major topic of research.

The use of a types module such as the one provided by the Napier88 system is crucial to the implementation of Octopus. The functionality provided by this module is used to implement wiring diagrams and to supply programs using Octopuses with type information.

The Octopus mechanism has been implemented using the Napier88 implementation platform. This platform has proven deficient as a vehicle for Octopus due to the Persistent Abstract Machine block retention architecture. A modification to this architecture has been proposed and an prototype implementation of it has been assessed with regard to performance. The hybrid architecture performs better than the PAM with respect to block retention. This property is especially important when first order information hiding is employed using first class procedures, as is common in Napier88 programs. The overall size of the data when boxing is employed is comparable in the two systems. Currently, all values are indiscriminately boxed, and there is considerable room for optimisation in this regard since local values that are not used by other procedures need not be boxed. These optimisations will further reduce the number of objects created within the Napier88 system and thus improve store utilisation and performance. These optimisations are under investigation at the time of writing.

Acknowledgements

This work is supported in part by the Defence Science and Technology Organisation of Australia through their assistance in the PIPE project, and by the Australian Research Council.

This paper also benefits from discussions with Francis Vaughan and Dave Hulse.

References

1. Connor, R. "The Napier Type-Checking Module", Persistent Programming Research Report 58, University of St. Andrews, 1988.

2. Connor, R. C. H., Brown, A. L., Cutts, Q. I., Dearle, A., Morrison, R. and Rosenberg, J. "Type Equivalence Checking in Persistent Object Systems", in *The Proceedings of the Fourth International Workshop on Persistent Object Systems*, Morgan-Kaufmann, Martha's Vineyard, Massachusetts, pp. 151-164, 1990.

3. Davie, A. J. T. and McNally, D. J. "PCASE - A Persistent Lazy Version of an SECD Machine", Research Report, The University of St. Andrews, CS/92/7, 1992.

4. Dearle, A. and Brown, A. L. "Safe Browsing in a Strongly Typed Persistent Environment", *The Computer Journal*, vol. 31, 6, pp. 540-545, 1988.

5. Farkas, A. and Dearle, A. "Octopus: A Reflective Language Mechanism for Object Manipulation", in *The Proceedings of the Fourth International Workshop on Database Programming Languages*, Springer-Verlag, New York City, 1993.

6. Farkas, A. M., Dearle, A., Kirby, G., Cutts, Q., Morrison, R. and Connor, R. "Persistent Program Construction through Browsing and User Gesture with some Typing", in *The Proceedings of the Fifth International Workshop on Persistent Object Systems*, Pisa, pp. 376-393, 1992.

7. Kirby, G. N. C. "Reflection and Hyper-Programming in Persistent Programming Systems", Ph.D. Thesis, University of St. Andrews, 1993.

8. Kirby, G. N. C., Connor, R. C. H., Cutts, Q. I., Dearle, A., Farkas, A. and Morrison, R. "Persistent Hyper-Programs", in *The Proceedings of the Fifth International Workshop on Persistent Object Systems*, Pisa, pp. 86-106, 1992.

9. Kirby, G. N. C. and Dearle, A. "An Adaptive Browser for Napier88", Research Report 90/16, University of St. Andrews, 1990.

10. Morrison, R., Atkinson, M. P., Brown, A. L. and Dearle, A. "On the Classification of Binding Mechanisms", *Information Processing Letters*, vol. 34, 2, pp. 51-55, 1990.

11. Morrison, R., Brown, A. L., Connor, R. and Dearle, A. "The Napier88 Reference Manual", University of St. Andrews, PPRR-77-89, 1989.

12. Peyton-Jones, S. "The implementation of functional languages", Prentice-Hall, 1987.

13. Stemple, D., Stanton, R. B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G., Fegaras, L., Cooper, R. L., Connor, R. C. H., Atkinson, M. P. and Alagic, S. "Type-Safe Linguistic Reflection: A Generator Technology", ESPRIT BRA Project 3070 FIDE Technical Report, FIDE/92/49, 1991.

14. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", in *the Proceedings of the 1st USENIX Conference on the Mach Operating System*, pp. 123-140, 1990.