Octopus:
A Reflective Language Mechanism for Object Manipulation

Alex Farkas
Department of Computing Science, University of Stirling
Scotland, FK9 4LA

Alan Dearle
Department of Computing Science, University of Stirling
Scotland, FK9 4LA

Abstract

A class of database programs exist which are required to operate over an infinite number of types; included in this class are object browsers and query tools. The types over which these programs operate cannot be enumerated statically. One solution to this problem is to provide a reflective language mechanism that permits the types of values to be abstracted over and the values manipulated in a type independent manner; this paper describes such a mechanism. The mechanism is called Octopus which is an acronym for Object Closure Transplantable to Other Persistent User Spaces. The essence of the technique 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. When manipulation is complete they may be *dropped* back into the value space, provided that they still conform to the language's type system. An additional feature of this technique, as the name suggests, is the ability to isolate portions of closures, and copy them to other locations. Partial closures may be *rewired*, possibly in a different context, using the meta level interface supplied by Octopus.

1 Introduction

1.1 Background

In most programming languages, programs and data form directed graphs with nodes consisting of arbitrary values, such as program fragments (i.e. procedures), records and arrays. Scalars form the leaf nodes of the graphs: they may be referenced but do not themselves reference other values. Figure 1.1.1 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 [17].

A database may be considered to be a collection of binding graphs such as the one shown above. Database applications consist of programs which construct

or navigate these graphs, in most cases the types of the nodes over which the program will operate are statically known by the database application programmer. For example, the programmer of an airline booking system will know that the nodes consist of seats, flights, people etc. In this case the programmer may write a, possibly strongly typed, program to navigate the graph performing some computation which may modify the graph.

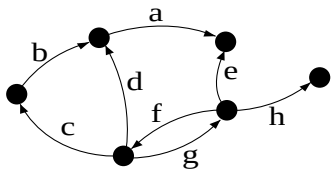

Figure 1.1.1: A conceptual view of programs and data structures.

Another class of database program exists in which the application programmer does not know the types of the data which may be encountered by the program. For example, consider an object browser that displays the contents of an arbitrary database to the user. Most programming languages permit an infinite number of types to be constructed, instances of which the browser may encounter. Clearly, in this case, some form of dynamic type checking must be provided. However, the problem is not entirely addressed by dynamic typing since the types of the values encountered by the browser cannot be textually enumerated.

One solution to these problems is to make use of some form of *linguistic reflection* [20]. Reflective systems permit their own structures to be altered from within. Using the notation of Stemple et al., programs are written in a representation of the language L called L_{rep} and have a universe of discourse, *Val*. For linguistic reflection to occur, L_{rep} must be a subset of *Val* or there must be a way of mapping a subset of *Val* into L_{rep}. Some mechanism must also be provided to map L_{rep} into *Val*; this is commonly provided by a function such as *eval* in Lisp.

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 allows the language to manipulate semantically meaningful constructs within that language and cause the compiler to perform computation over these constructs. This allows the user to write generic code that could not otherwise be written. This technique is akin to having a macro processor for the host language which is context sensitive and has the ability to manipulate context. The reader is referred to [20] for more details of this important language mechanism.

When dynamic linguistic reflection is employed, new programs are constructed dynamically and introduced to the running program. For example, when the PS-algol [7] and Napier88 [15] 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. This new program is cached in the closure of the browser for future use. This technique requires two facilities in the programming language. Firstly, since the browser program can take any data type as a parameter, some dynamic type checking is required. This may be

satisfied using a type such as *dynamic* in Quest [4], or *any* in Napier88 [18]. The second requirement is that the reflective part of the language has the ability to deliver the type of the value supplied to the browser. In Napier88 this is achieved through the use of a special function which delivers a representation of the type of a value encapsulated in an *any*. Making use of such reflective techniques can be expensive. For example, the construction of generic tools is complicated by the need to construct programs which have the ability to construct and compile further programs to deal with unknown types.

1.2 Octopus

This paper describes a new dynamic linguistically reflective mechanism called Octopus. This mechanism 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. In essence, 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. The essence of the technique is to allow values from the programming language value space to be *hoisted* up to the meta level and manipulated in ways which the programming language would not otherwise permit. When manipulation is complete, they may be *dropped* back into the value space, provided that they still conform to the language's type system.

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, a hoisting procedure may be called to convert a value into an Octopus. 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 using the same set of operations.

The Octopus model provides the ability to *cut* and *rewire* the bindings within a hoisted value. When a value from the value space is hoisted up to the meta level, all of its bindings are treated as hooks from the hoisted value to the bound values. A binding is cut by detaching the hook from the bound value, and is rewired by attaching the hook to another value of the same type; it is not possible to rewire values of an incompatible type. Neither is it possible to drop the hoisted value back into the value space until all bindings are correctly rewired.

The Octopus model facilitates the distribution of object closures by allowing partial closures to be exported to other persistent stores. Upon arrival, the Octopus may be rewired and dropped into the value space. The main advantage of this technique is that it removes the need to unnecessarily duplicate and transport values which may already be present at the destination. The model is targeted at persistent systems in which software components, such

as window manager systems or compilers, are often common to many persistent stores. The model allows values to be detached from such common values and rewired to the equivalent values in another store.

This paper describes the Octopus model and a number of applications of the model. The language used to implement the model is the persistent programming language Napier88, and the applications are described in terms of simple Napier88 examples.

The paper is structured as follows: Section 2 describes the Octopus model, Sections 3, 4, 5 and 6 describe, in detail, some applications of the model, while Section 7 offers some conclusions.

2 The Octopus Model

Two special operations exist which allow a value to be hoisted to and dropped from an Octopus. Type declarations for the two operations are shown in Figure 2.1.

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

Figure 2.1: The Octopus hoist and drop operations.

The first operation, *coerceToOctopus*, hoists a value up into an Octopus. Using the notation of Stemple et al. this is a mapping from *Val* to L_{rep}. In the Octopus model, L_{rep} is a subset of *Val* and consequently the hoisted value is still part of the value space. The other operation, *coerceFromOctopus*, first checks to ensure that all bindings in the Octopus are resolved before extracting the encapsulated value. If any of the bindings are unresolved, the Octopus is returned unchanged. This operation maps from L_{rep} to *Val*.

The *coerceToOctopus* operation produces an Octopus which provides a view of the hoisted value. The operations provided by the Octopus act on the original value rather than a copy of the value, and, in particular, manipulation of the encapsulated value is performed in place. Similarly, the *coerceFromOctopus* operation causes the actual value encapsulated in the Octopus to be dropped into the language value space.

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. This functionality is delivered by Napier88 through 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.

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

Figure 2.2: The structure of an Octopus.

An Octopus is a set of three operations which allow the bindings within a value to be examined and manipulated. These operations are 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 corresponding type declaration for an Octopus is shown in Figure 2.2.

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. A complete set of selector, constructor, equivalence and iterator functions are provided by the Napier88 system that operate on this representation [5]. Thus, using these functions, the programmer can obtain any required information about the type.

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 [9], [13] and [14]. 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. The model of source representation is the topic of related research and will not be discussed further in this paper.

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, the procedure may be used to display a binding's value. Bindings are also represented as a package of functions, and are described below.

2.1 Bindings

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

Figure 2.1.1: The representation of a binding.

Each binding is represented by six operations, as shown by the corresponding type declaration in Figure 2.1.1. The operations on bindings behave as follows:

cut When applied, this operation causes the associated binding to be dissolved. As described in the introduction, 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.

```
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

type example is structure( a : int ; b : string ; c : bool )
let myrecord = example( 1,"hello",true )

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

Figure 2.1.2: A program to display the type of a value.

```
a : int
b : string
c : unknown type
```

Figure 2.1.3: Output of the program shown in Figure 2.1.2.

A simple example of the use of the Octopus operations is shown in Figure 2.1.2 which displays the types of the fields of the record denoted by *myrecord*. The output of this program is shown in Figure 2.1.3. 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. This is achieved by obtaining the type of each binding using the *getType* operation of the binding and checking for type representation equality using the *EqualType* operation provided by the Napier88 system. As presented in this example, *scanner* only has knowledge of two types, integer and string. However a more sophisticated version of this procedure could be written using the *getType* operation provided by Octopuses to display the type of an arbitrary value.

3 Viewing

A common activity in persistent programming and database environments is browsing. Object browsers such as those described in [7], [9] and [15] are used for debugging data structures and applications, for viewing values and for discovering reusable components in the object repository [3]. This is achieved by allowing arbitrary data structures to be examined and by displaying graphical representations of values in the form of menus connected by arrows. Thus, the values and the relationships between objects may be graphically displayed.

As described above, some persistent object browsers generate and compile code on demand at run-time. This technique can be expensive due to construction and compilation overheads. Browsers which avoid this overhead have been constructed by making use of *magic* functions. Magic functions allow the programmer to perform operations below programming language type system level. Typical of these operations are assignment, the construction of new values and the ability to obtain type information. These functions are not generally

available to the Napier88 programmer since they fundamentally compromise the type system of the programming language.

The Octopus model simplifies the construction of browsing tools by removing the need to compile new browsing routines, whilst retaining the security offered by the type system of the language.

As the Octopus model provides a uniform interface to values, the code for interrogating values becomes more succinct and generic than using the programming language alone. For example, consider the program in Figure 3.1.1 which presents the user with menus for browsing an arbitrary object closure.

The program shows the declaration of a procedure called *browse* which takes as its parameter an Octopus. The procedure first creates an empty menu by calling a procedure called *createMenu*. Next, a procedure called *makeMenuEntry* is declared which, when applied to a binding, causes an entry for the binding to be inserted into the menu. Each menu entry consists of two parts: a name that is displayed to the user and a procedure which is executed when the entry is selected. An entry is inserted into the menu as follows: firstly, the name of the bound value is obtained using *getName*. Next, the procedure to be executed, *action*, is declared, which extracts the bound value using *get* and injects it into an Octopus before passing it to a recursive call of the browser. Finally, the bound value name and *action* are inserted into the menu.

```
rec let browse = proc( octopus : Octopus )
begin
        !** create new, empty menu
        let menu = createMenu()

        !** procedure to add a binding to a menu
        let makeMenuEntry = proc( b : Binding )
        begin
                !** get name of bound value
                let entry = b( getName )()

                !** the procedure to call when this menu entry is selected
                let action = proc(); browse( coerceToOctopus( b( get )() ) )

                !** add binding to menu
                menu( addEntry )( entry,action )
        end

        octopus( scan )( makeMenuEntry )
        menu( display )()
end
```

Figure 3.1.1: A simple menu-style browser.

The menu to represent the value is constructed by applying the *scan* operation of the Octopus with the *makeMenuEntry* procedure as a parameter. The *scan* operation applies *makeMenuEntry* to each binding in the Octopus and causes an entry for each binding to be placed in the menu. This menu is then displayed on the screen by calling the menu *display* procedure. When a menu entry is

selected by the user, the appropriate action procedure is called which causes the browser to be recursively applied to the corresponding bound value.

The browser in Figure 3.1.1 is able to traverse all values in the value space due to the uniform interface provided by Octopuses. Thus, the amount of code required to construct the browser is reduced, and the need to dynamically adapt to values whose types have not been encountered previously by the browser is removed.

4 Querying Complex Objects

In the previous section, it was shown that the Octopus model may be used to provide a uniform view of values. The techniques used to browse an object closure may be applied in order to perform a query over an arbitrarily complex object closure. For example, suppose all of the bindings with the name *balance*, of type integer and with value greater than 100 within a particular closure are required. In a language such as SQL [12], dedicated to performing database operations, this may be achieved by a query such as the one shown below.

```
select object from database
where object.balance > 100

let resultList = createList()
let visitedList = createList()

rec let query = proc( octopus : Octopus )
if ~visitedAlready( octopus ) do
begin
        !** The function to be passed to scan.
        let check = proc( b : binding )
        begin
                if  b( getName )() = "balance" and
                    isType( b( getType )(),INT ) then
                begin
                    project b( get )() as i onto
                        int       :   if i > 100 do
                                      addToResultList( octopus )
                        default   :   {}
                end
                else
                    query( coerceToOctopus( b( get )() ) )
        end
        !** Query body
        addToVisitedList( octopus )
        octopus( scan )( check )
end
```

Figure 4.1: Querying an object closure.

In a programming language such as Napier88 this kind of query could be written in one of two ways. Firstly, if the types of the values over which the query was to be performed were known, a statically typed program could be written. This

program could not be used on data structures of any other type since it has type information, such as field labels, encoded into it. Parametric polymorphism yields no extra power: a polymorphic query function could be written but it would be required to be parameterised by selector functions for each field of each type, and there may be an unbounded number of these. However, a more general purpose program could be written which dynamically examined the types of the values to which it was applied. In order to cater for an unbounded number of types, this program would be required to use the reflective techniques described earlier.

In all of the above cases not all the values in a closure may be accessed. Those values encapsulated in procedures and abstract data types remain hidden. Using Octopus, it is possible to construct a program which performs a query over an arbitrarily typed set of values. Furthermore the Octopus mechanism permits all the values, encapsulated or otherwise, within a closure to be examined. This ability is somewhat controversial and is discussed in the conclusions. The program shown in Figure 4.1 illustrates how a database query such as the one shown above may be implemented using the Octopus mechanism.

The first two lines of the program create two empty lists. The first list, *resultList*, is used to collect all Octopuses within an object closure which contain a binding to a value with name *balance*, of type integer and with magnitude greater than 100. The second, *visitedList*, is used to record those Octopuses in the closure which have already been visited by the query.

If a given Octopus has not been visited, the *query* procedure adds it to the visited list and then examines its bindings by supplying a procedure to the Octopus's *scan* operation. The procedure *check* examines the bound value, and if it has the name *balance*, is of type integer with a magnitude greater than 100, the Octopus containing this binding is inserted into the *resultList*. Otherwise, the bound value is converted into an Octopus and supplied to the *query* procedure. In this manner, a depth first query of the object closure is performed.

4.1 High level query abstractions

The selection criteria of the query shown in Figure 4.1 is domain specific, it only finds integer values called *balance* whose values are greater than 100. It is possible to abstract over the *query* procedure by allowing the selection criteria to be passed to the query as a parameter.

To illustrate this, consider the query shown in Figure 4.1.1. A procedure called *Find* is declared, which takes a selection predicate as one of its parameters. Octopuses associated with bindings which satisfy the predicate are collected and returned.

The procedure *Find*, is one of a number of high level abstractions over Octopuses. Figure 4.1.2 contains a list of other operators we have found to be useful. *Find* has already been described, *Apply* applies the procedure *action* to every binding in the closure. *CondApply* is an extension of *Apply* that applies the procedure *action* to those bindings which satisfy *pred*. The last operator, *Fold*, is polymorphic, and applies the function *map* to all bindings in the closure. Each call of *map* returns a value which is supplied to the collector procedure *collect*. The result of the final call to *collect* is returned by *Fold*. Each of the previous operations may be constructed in terms of *Fold*, but are implemented separately as an optimisation. It is the subject of current research to find a useful set of

such operations which may be used to implement a number of tasks such as the ones described in this paper.

```
let visitedList = createList()

rec let Find = proc(   octopus : Octopus ;
                       select : proc( Binding → bool ) → OctopusList )
if ~visitedAlready( octopus ) do
begin
    let resultList = createList()
    !** The function to be passed to scan.
    let check = proc( b : binding )
    begin
        if  select( b ) then
            addToResultList( octopus )
        else
            concat(   resultList,
                      Find( coerceToOctopus( b( get )() ),
                          select,action ) )
    end
    !** Find body
    addToVisitedList( octopus )
    octopus( scan )( check )
    resultList
end
```

Figure 4.1.1: A general querying procedure to return a list of Octopuses.

```
Find      :  proc(   octopus : Octopus ;
                     pred : proc( Binding → bool ) → OctopusList )

Apply     :  proc(   octopus : Octopus ; action : proc( Binding ) )

CondApply:   proc(   octopus : Octopus ; pred : proc( Binding → bool ) ;
                     action : proc( Binding ) )

Fold      :  proc[ t,l ](   octopus : Octopus ; map : proc( Binding → t ) ;
                            collect : proc( t, l → l ) → l )
```

Figure 4.1.2: High level operators.

It has been shown that the Octopus mechanism may be used to implement arbitrary queries over the value space. The approach is to provide some low level, yet powerful, operations on top of which queries may be implemented. In this way, many query languages and paradigms may be provided by a single database environment. This is in contrast to other database programming languages such as Galileo [1] and DBPL [16] which provide a fixed set of higher level query operations as intrinsic features of the language.

One of the drawbacks of this approach is that there is less opportunity for optimisation: each Octopus provides a uniform interface to an arbitrary value, so optimisations which are dependent on the type of a value are impossible to make. Furthermore, the fact that procedures passed to the operations have unknown side effects makes optimisation almost impossible.

Lastly, the composition of various library routines makes no allowances for the possible algebraic optimisations of a query which may exist. This opportunity may be regained by constructing query languages which are translated into operations in terms of Octopuses.

5 Evolution of Programs and Data

The technique of programming enabled by Octopuses may also be used to support program and data evolution. Data and programs may be evolved from one form into another by cutting old components and wiring in new ones. To illustrate this, consider a database of parts [2] with the type shown in Figure 5.1.

```
type Part is structure(   name     :  string ;
                         number   :  int ;
                         describe :  proc() )
```

Figure 5.1: Type of data held in parts database.

Each part in the database is represented by a record of three components: a name, a unique number and a procedure which, when applied, displays a description of that part. Each instance of *Part* is bound, via the name *describe*, to the same procedure, *display*, which is encapsulated within *describe*. To clarify, a constructor function for parts is shown in Figure 5.2 below.

```
let display = proc( s : string ); !** a procedure to display a string

let createPart = proc( name,description : string ; number : int → Part )
      !** Return a new record. Part is used as a constructor here.
      Part(  name,                              !** name
             number,                            !** number
             proc(); display( description ) )   !** describe
```

Figure 5.2: The *Part* constructor function.

Suppose that a parts database has been populated with parts created using the *createPart* function, and that the *display* function has been found to be erroneous or inefficient. In order for the parts database to make use of a new display function, all instances of *Part* bound to *display* need to be located and updated.

Using the Octopus model, a *Part* record may be hoisted into an Octopus, the binding to the old *display* procedure cut and a binding to the new *display* procedure established. This process is illustrated by the program in Figure 5.3.

In this program, a predicate, *select*, is declared which examines a binding to see if it has the name *display* and if the associated value is a procedure which takes a single string as a parameter. Next, an update procedure, *updateDisplay*, is declared which causes bindings to the old *display* procedure to be updated. It achieves this by cutting the original binding and attaching the *newDisplay* procedure. Finally, the program locates and updates all values in the parts database which are bound to the *display* procedure using the *CondApply* operator described earlier.

```
let newDisplay = proc( s : string ); ... !** a new display procedure.

!** Define the selection criteria.
let select = proc( b : Binding → bool )
        b( getName )() = "display" and
        EqualType( b( getType )(),typeRep( "proc( string )" ) )

!** Update the binding.
let updateDisplay = proc( b : Binding )
begin
        !** First, cut the binding to the old display procedure.
        let ok := b( cut )()
        !** Next, bind in the new display procedure.
        ok := b( add )( newDisplay )
end

!** Update all components which are bound to display.
CondApply( coerceToOctopus( partsDB ),select,updateDisplay )
```

Figure 5.3: Evolving the parts database.

6 Distribution of Complex Object Closures

The task of copying arbitrary object closures from one persistent store to another is complex since, in practice, object closures tend to span all (or a large proportion) of the objects in an object store. One way of tackling this problem is to permit parts of an object closure to be isolated, by cutting a number of bindings, so that the object closure can be partially copied and rewired in another context. This was the original motivation for the Octopus model and the task from which its name derives.

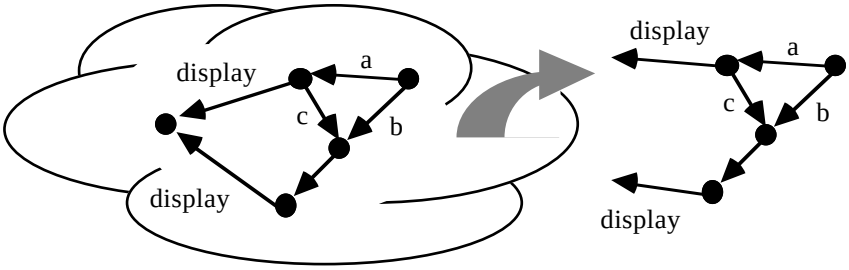

Figure 6.1: Distributing the parts database.

To illustrate this technique, suppose that the parts database is to be installed in another persistent store. Various components of the database are bound to the *display* procedure, and this *display* procedure is common to all persistent stores. In order to avoid unnecessary copying, the application is converted into an Octopus, and the bindings to *display* are cut. The resulting Octopus is then transmitted to a destination persistent store and rewired. Figure 6.1 shows a simplified, conceptual view of this process.

When the transmission is complete, the database may not be dropped back into the value space until the closure is wired to the necessary components, i.e.

the unresolved *display* bindings must be rewired to the *display* procedure in the new persistent store. Once the rewiring is complete, the installed version of the parts database has the same state as the original database.

Rewiring may be achieved using a program sent to the other store along with the closure to be rewired. This program is equivalent to an installation script in the Macintosh environment [11]. Figure 6.2 shows an installation procedure which may be transmitted to the destination persistent store in order to automatically rewire and reinstall the parts database.

The *rewire* procedure is constructed in a similar way to the procedure in Figure 5.3, used to evolve the parts database. Firstly, a selection procedure, *select*, is declared, followed by a computation which locates the *display* procedure in the destination persistent store. A procedure, *rebind*, is declared which binds the procedure *localDisplay* to the closure. *CondApply* is used to establish the rebinding process. Finally, the Octopus is dropped back into the programming language value space and placed in the persistent store.

To allow transmission, two operations *exportOctopus* and *importOctopus* are provided. The mode of transmission may be a remote procedure call such as the one described in [8], or simply the placement of the partial closure in a file.

```
let rewire = proc( octopus : Octopus )
begin
    !** Define the selection criteria.
    let select = proc( b : Binding → bool )
        b( getName )() = "display" ...

    !** Locate the display procedure in the destination persistent store.
    let localDisplay = ...

    let rebind = proc( b : Binding )
    begin
        let ok = b( add )( localDisplay )
        if ~ok do error( ... )
    end

    !** Rewire all components which are bound to display.
    CondApply( octopus,select,rebind )

    !** Place parts database in new store.
    in PS() let application = coerceFromOctopus( octopus )
end
```

Figure 6.2: A procedure to rewire and install the parts database.

7 Conclusions

The Octopus model has been implemented in Napier88. In the process of implementing the model, the architecture of the Napier88 system was modified in a number of ways to provide support for some of the features of Octopuses.

The primary changes were to make use of boxed representations [19] for values and to combine this with the idea of flat static environments [6]. These changes, whilst not affecting the language itself, allowed the cut and rewire

operations of Octopuses to be implemented as described in this paper. One of the additional benefits of this architecture is an improvement in storage utilisation, by decreasing the number of retained objects within closures. However, the price paid is an additional dereference on values which would have remained unboxed in the original Napier88 architecture. A more detailed description of this architecture is the topic of another paper.

In the current prototype of the system described in this paper, the source code obtained from an Octopus is a linear textual representation. Our intention is to extend the source code representation to support the features offered by hyper-programming [13, 14]. In addition, by allowing partially resolved hyper-programs to exist, a method of template programming [10] may be developed. These techniques appear to be complementary to the ideas presented in this paper.

In this paper a mechanism which provides a uniform abstract interface to values of any type has been presented. The Octopus approach is to provide a few simple, but powerful reflective mechanisms in the language and to use these to construct higher level tools. These relatively simple mechanisms provide 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, it has been shown how the Octopus mechanism may be used to browse and query graphs of values of arbitrary data types.

Perhaps the most controversial aspect of Octopus is the ability to examine bindings encapsulated within functions, procedures and abstract data types. This undoubtably breaks the encapsulation and information hiding associated with these language constructs. This raises a philosophical argument as to whether this information should be available to (meta) language mechanisms. Two arguments favour this approach, one is the fact that this mechanism permits programs to be written that could not otherwise be written. The other is the fact that the protection and encapsulation afforded by these mechanisms is not compromised unless the reflective constructs are used. No doubt we will be judged in the fullness of time.

Acknowledgments

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.

References

1. Albano, A., Cardelli, L. and Orsini, R. “Galileo: A Strongly Typed, Interactive Conceptual Language”, ACM Transactions on Database Systems, vol 10, 2, pp. 230-260, 1985.
2. Atkinson, M. P. “Malcolm’s Famous Parts Example”, Personal Communication, 1887.
3. Bachman, C. W. “The Programmer as Navigator”, Turing Award Lecture, *in Communications of the ACM*, vol 16, 11, pp. 653-658, 1973.
4. Cardelli, L. “Typeful Programming”, Research Report 45, DIGITAL Systems Research Center, 1989.

5. 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 Proceedings of the 4th International Workshop on Persistent Object Systems*, Morgan-Kaufmann, Martha's Vineyard, Massachusetts, pp. 151-164, 1990.
6. Davie, A. J. T. and McNally, D. J. "PCASE - A Persistent Lazy Version of an SECD Machine", Research Report, University of St. Andrews, CS/92/7, 1992.
7. Dearle, A. and Brown, A. L. "Safe Browsing in a Strongly Typed Persistent Environment", *The Computer Journal*, vol 31, 6, pp. 540-545, 1988.
8. Dearle, A., Rosenberg, J. and Vaughan, F. "A Remote Execution Mechanism for Distributed Homogeneous Stable Stores", *in Proceedings of the Third International Workshop on Database Programming Languages*, Morgan Kauffman, pp. 125-138, 1991.
9. 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 Proceedings of the Fifth International Workshop on Persistent Object Systems*, Pisa, pp. 376-393, 1992.
10. Futatsugi, K., Goguen, J., Meseguer, J. and Okada, K. "Parameterized Programming in OBJ2", *in Proceedings of the Ninth International Conference of Software Engineering*, pp. 51-60, 1987.
11. Apple Computer Inc., "Inside Macintosh", Addison-Wesley, 1986.
12. American National Standards Institute, "Database Language SQL", New York, 1986.
13. Kirby, G. N. C. "Reflection and Hyper-Programming in Persistent Programming Systems", Ph.D. Thesis, University of St. Andrews, 1993.
14. Kirby, G. N. C., Connor, R. C. H., Cutts, Q. I., Dearle, A., Farkas, A. and Morrison, R. "Persistent Hyper-Programs", *in Proceedings of the Fifth International Workshop on Persistent Object Systems*, Pisa, pp. 86-106, 1992.
15. Kirby, G. N. C. and Dearle, A. "An Adaptive Browser for Napier88", Research Report, University of St. Andrews, 90/16, 1990.
16. Matthes, F. and Schmidt, J. "The Type System of DBPL", *in Proceedings of the 2nd International Workshop on Database Programming Languages*, Oregon, pp. 219-225, 1989.
17. 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.
18. Morrison, R., Brown, A. L., Connor, R. and Dearle, A. "The Napier88 Reference Manual", University of St. Andrews, PPRR-77-89, 1989.
19. Peyton-Jones, S. "The implementation of functional languages", Prentice-Hall, 1987.
20. 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.