Changing Persistent Applications

Alex Farkas and Alan Dearle

{alex,al}@cs.adelaide.edu.au
Department of Computer Science, University of Adelaide
Adelaide, Australia

Abstract

During the lifetime of an application, the objects and bindings in a persistent store may require modification in order to fix bugs or incorporate changes. Two mechanisms, Octopus and Nodules, supporting the evolution of persistent applications are presented. The first, Octopus permits code and data values to be evolved, even if they are encapsulated. Type evolution is addressed by the separation of type information from the executable code. In many cases type evolution is possible, without the expense of total or partial system recompilation. Nodules are a complementary mechanism to Octopus in that they allow generic templates to be defined independently of any referencing environment. Nodules may be specialised in order to yield instances by binding them to values and types. When combined into a single system, Nodules and Octopus enable a rich collection of information about the structure and state of applications to be maintained and made available to programmers not only during the construction phase, but during the entire lifetime of applications.

1 Introduction

Persistent applications consist of graphs in which the nodes are data objects and the arcs are bindings between them. In a persistent programming language which supports first class procedures[1], such as Napier88 [10], the objects in the store include procedures with bindings to data, and data with bindings to procedures. Persistent systems support incremental construction [4] and component reuse [9] by allowing components to be created, stored in the persistent object store and bindings between them established. Bindings may be established at different times: during program construction, at program compile time and when the program is executing [7].

[1] Throughout this paper, the terms program, procedure and functions are used synonymously.

```
type Part is structure( name : string ; id,quantity : int )

!** The database is encapsulated within the partsDatabase procedure whose
!** parameters are a command, a part name and a quantity.
let partsDatabase =
begin
    let database = !** generate a b-tree for storing Parts.

    !** Declare three procedures which are bound to and manipulate
    !** the database.
    let create = proc( partName : string ; amount : int )
    begin
        let newId = ... !** generate a new unique identifier.
        let newPart = Part( partName,newId,amount )
        database( enter )( newPart )
    end

    let update = proc( partName : string ; amount : int )
    begin
        let apart = database( lookup )( partName )
        !** Next line is an error, rhs should be apart( quantity ) + amount
        apart( quantity ) := apart( id ) + amount
    end

    let display = proc( partName : string )
    use PS() with IO in
    use IO with writeInt : proc( int ) ; writeString : proc( string ) in
    begin
        let apart = database( lookup )( partName )
        !** call a procedure to display the number of parts.
        writeString( apart( name ) )
        writeInt( apart( id ) )
        writeInt( apart( quantity ) )
    end

    !** Define the partsDatabase procedure which is bound to the utility
    !** procedures.  This procedure is returned as the result of the
    !** computation between the outermost begin and end.
    proc( command,partName : string ; amount : int )
    begin
        case command of
        "CREATE"   :  create( partName,amount )
        "BUY"      :  update( partName,amount )
        "SELL"     :  update( partName,–amount )
        "PRINT"    :  display( partName )
        default    :  !** error ...
    end
end
```

Figure 1: A simple program creating a database of *Part*s.

During the lifetime of an application, the objects and bindings in the object store may require modification in order to fix bugs or incorporate changes. In general, evolution involves traversal of the object graph in order to modify, create and delete objects and bindings. This may be achieved through the use of a persistent store browser [3]: a general purpose tool which traverses object graphs applying a function to the objects it encounters. This is analogous to providing a generic *map* operation over an object graph. For example, a browser may be used to find and update all the instances of a given type in a database. The use of browsers to aid evolution has been proven in the context of Napier88 persistent object stores. However, a number of limitations have been encountered, namely:

1. Encapsulated values in the object graph cannot be reached using a browser, since it is impossible to browse through functional interfaces.
2. Evolved code cannot be bound to existing data and evolved data cannot be bound to existing code since the existing code and data cannot be named or referenced.
3. The lack of support for type evolution.

Encapsulation plays an important role in the construction of many applications. Abstract Data Types (ADTs) [8] and the Object-oriented design methodology [1] rely heavily on encapsulation in order to hide the underlying data structures of an application. Such information hiding is often cited as an aid to system evolution since the implementation of an abstract type can be changed without affecting the programs that make use of that abstract type. However, abstract types can cause difficulties when used as a mechanism to encapsulate persistent objects – when the source of an abstract type is recompiled and a new abstract type installed, the persistent state will be lost. If the interface to the abstract type is complete, it may be possible to extract all the encapsulated data from the old instance and place it in the new instance with no loss of information. However, this is not always the case, as in the *partsDatabase* example shown in Figure 1 in which the data is encapsulated but cannot be accessed through the (degenerate) functional interface.

System evolution consists of three activities: evolving data, evolving code and evolving types. Evolution of data is perhaps the simplest activity: provided that the data structure is not encapsulated and therefore reachable, a browser can traverse it and create a new isomorphic data structure. However, in order to ensure referential integrity, all the references to the old data structure must be found and consistently updated. This is also possible using a browser (if a little time consuming). However, bindings to data structures encapsulated within the closures of functions, present a problem; for example, the binding to *database* from *create* in Figure 1. There is no method of finding these bindings using existing browser technology. A similar problem exists with evolved code – if referential integrity is to be preserved, the evolved version of the code should contain the same bindings as the old. Clearly some general purpose mechanism is required that is capable of reaching all data regardless of whether it is bound or encapsulated.

Applications generally make use of an arbitrary set of types. As an application evolves, these types may undergo changes which effect the application in two primary ways: firstly, programs which manipulate values of

the evolved types may also need to be evolved to reflect the change in type. Secondly, programs which perform dynamic type checking and create instances of the types must be updated.

In the remainder of this paper, we present evolutionary techniques which address the problems discussed above. The mechanisms are based on the principle that all mutable information should be removed from the code and stored separately. The technique also provides a way in which generic executable code may be obtained and reused with different values and types. A mechanism called a *Nodule* is presented that allows such unbound code to be specified and bound at a later time.

This paper is structured as follows. Section 2 describes Octopus, a mechanism for interrogating and manipulating bindings, Section 3 describes a mechanism for propagating type changes through existing programs, Section 4 describes Nodules and section 5 describes how these mechanisms form a part of the programming environment.

2 Octopus

A mechanism called Octopus is described in [5] and [6] which allows all bindings in an arbitrary graph of objects to be examined and manipulated. In essence, Octopus provides a uniform viewing mechanism with which values and bindings of any type may be viewed and manipulated using the same set of operations. The mechanism allows values to be hoisted up to a meta level and manipulated in ways which the programming language would not otherwise permit. When manipulation is complete, values may be dropped back into the value space provided they still conform to the language's type system.

An Octopus is a set of three operations which allow the bindings within a value to be examined and manipulated. The corresponding type declaration for an Octopus is shown in Figure 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 that operate on this representation are provided by the Napier88 system [2]. Using these functions, the programmer can obtain any information about the type.

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

Figure 2: The structure of an Octopus.

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 [7]. 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 *getBinding* operation returns the binding associated with the given name if one exists. 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.

The Octopus model provides the ability to *cut* and *rewire* the bindings within hoisted values. 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.

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

Figure 3: The representation of a binding.

A binding is represented as a package of six operations, as shown in Figure 3. The operations on bindings behave as follows:

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	permits 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	returns the current value of the binding. If the binding is unresolved, a fail value is returned.
resolved	returns *true* if the binding is in a resolved state and *false* otherwise.
getType	returns a representation of the type of the corresponding bound value.
getName	returns the name of the bound value.

2.1 Implementation of Octopus

Octopus has the ability to traverse and manipulate arbitrary object graphs, including encapsulated values. The ability of Octopus to access encapsulated bindings requires special attention in the implementation architecture. To illustrate this, we will describe how the *partsDatabase* application is represented and how Octopus manipulates this representation. Readers are referred to [5] for a more detailed description of the Octopus architecture.

In systems supporting Octopus, all bindings encapsulated in procedures are stored separately from executable code. All procedural values are represented by two entities: a structure containing only executable code, known as a *code vector*, and a structure containing the encapsulated bindings, known as an

environment vector. Together, a code vector and environment vector form the *closure* of a procedure. The *partsDatabase* procedure is represented as shown in Figure 4.

On the left hand side of the figure, the closure for the *partsDatabase* procedure is shown. The environment vector contains bindings to each of the three operations *create*, *update* and *display*. Similarly, the environment vector of each operation contains a binding to the database.

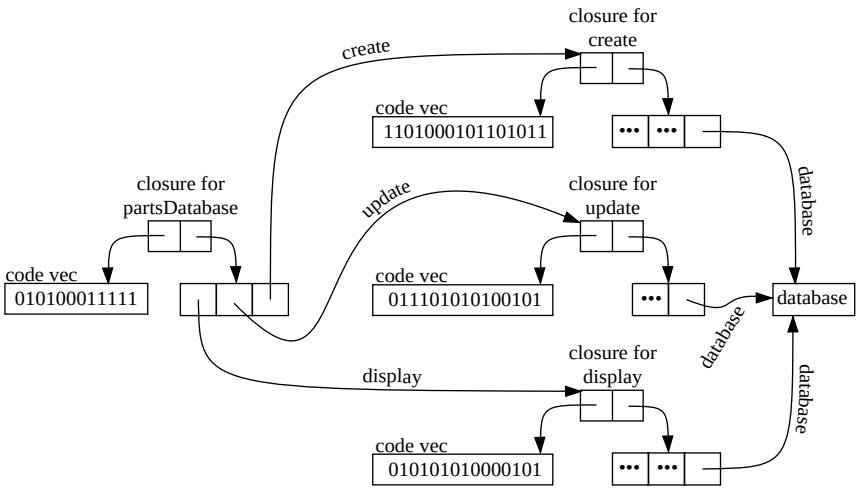

Figure 4: Conceptual view of the *partsDatabase* application.

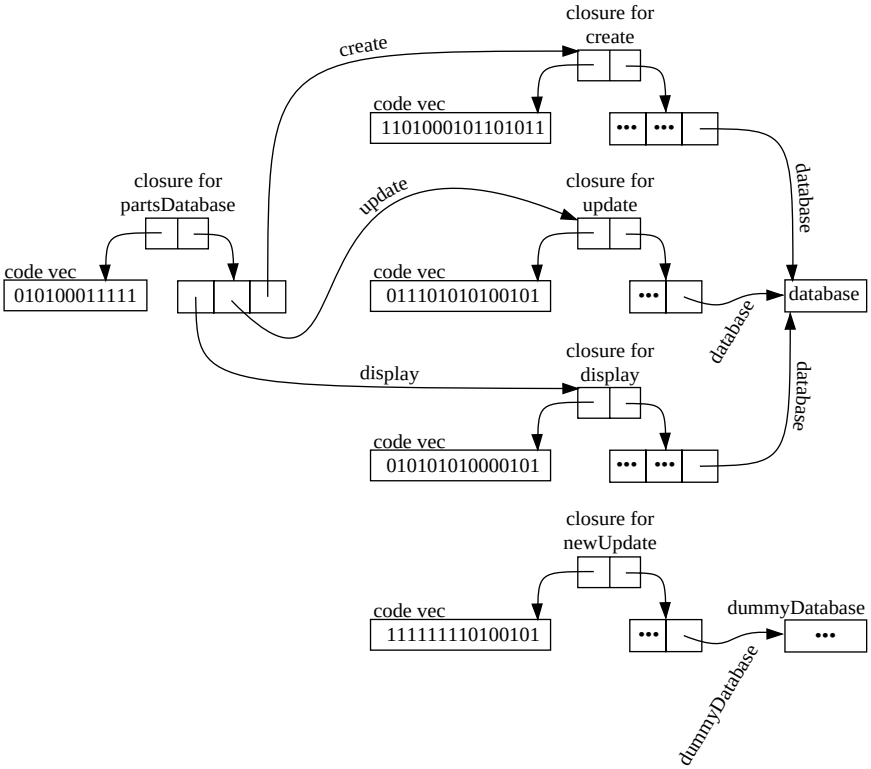

Figure 5: View of *newUpdate*, the replacement procedure for *update*.

Octopus may be used to traverse object graphs, access encapsulated values, and perform rebinding of new values. For example, consider the *update* procedure used by *partsDatabase*, *update* is faulty as it increments the *id* field of a part instead of the *quantity* field, so it needs to be replaced. There are two steps involved in making this change:

1. The new procedure must be created and bound to *database*, and
2. the *partsDatabase* application must be bound to the new procedure.

In order to bind the new procedure to *database*, it must be located. This may be achieved by traversing the object graph of the *partsDatabase* application using the Octopus operations. Once located, the new procedure may be bound to *database*. This could be achieved using the Octopus *add* operation or, since the location containing the database used by the new procedure is known, by assignment. Finally the rewired procedure may be bound into the *partsDatabase* application using the Octopus *add* operation. This process is illustrated in Figures 5 and 6.

In Figure 5, a procedure *newUpdate* has been created to be used as the replacement for *update*. Initially, *newUpdate* is bound to a dummy database; this is to be replaced by a binding to the database used in *partsDatabase*.

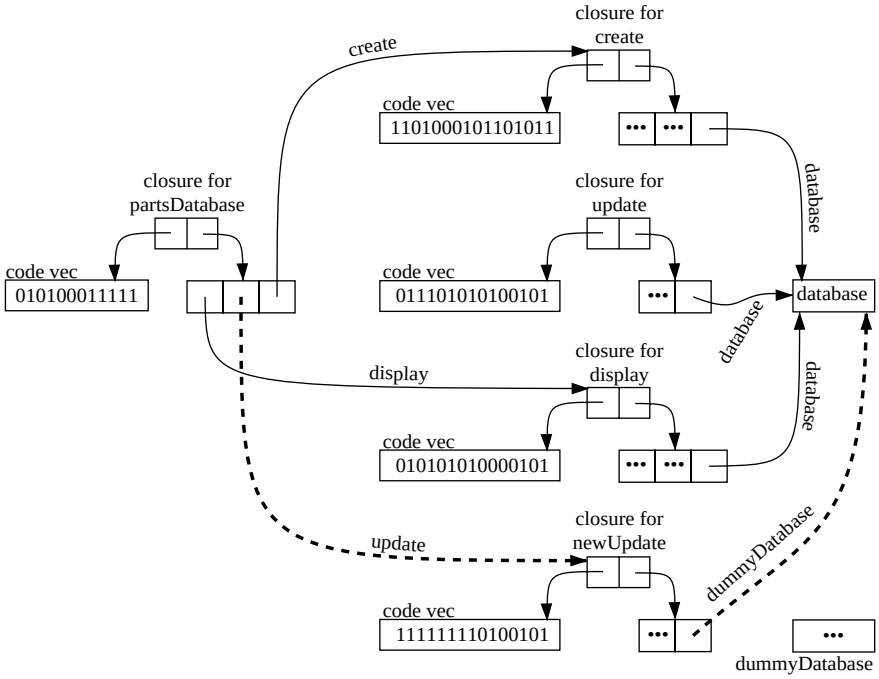

Figure 6: View of *partsDatabase* after wiring in *newUpdate*.

Figure 6 shows the application after *newUpdate* has been bound to the database and *partsDatabase* to *newUpdate*. There are no other bindings to the old *update* procedure or the dummy database and these will be garbage collected by the system. The Napier88-like pseudo code in Figure 7 shows how these steps are achieved.

The first line of the program constructs a dummy instance of a database. Next, a procedure *newUpdate* intended as a replacement for the faulty *update* procedure is declared; this procedure is bound to *dummyDatabase*. Then, *newUpdate* and *partsDatabase* are hoisted into Octopuses using the special function *coerceToOctopus*. The *getBinding* operation of the hoisted database is used to obtain a binding to the original *update* procedure, which is then hoisted. Similarly, the *getBinding* operation of the hoisted original *update* procedure is used to obtain the binding to the old database, which is then assigned to *dummyDatabase*. Finally, the binding to the old *update* procedure is cut and the *newUpdate* procedure wired in.

Octopus allows arbitrary object graphs to be interrogated, but only allows values within the graphs to be manipulated or replaced. No support is given to enable changes in schema to be propagated through the graphs. For example, if the type *Part* in the parts database is extended to include the year in which a part was manufactured, then three activities need to occur. Firstly, the source code for the application needs to be modified to use the new type. Secondly, the programs need to be recompiled and lastly, the values in the database need to be evolved to be consistent with the new type. This may be achieved by traversing the database using Octopus and constructing an isomorphic graph of values

which include the new field. The new database may then be wired into the new application.

```
let dummyDatabase := ... !** declare a dummy value.

!** The new version of update
let newUpdate = proc( partName : string ; amount : int )
begin
        let part = dummyDatabase( lookup )( partName )
        part( quantity ) := part( quantity ) + amount
end

let hoistedDb               =  coerceToOctopus( partsDatabase )
let oldUpdateBinding        =  hoistedDb( getBinding )( "update" )
let oldUpdate               =  oldUpdateBinding( get )()
let hoistedOldUpdate        =  coerceToOctopus( oldUpdate )
let oldDbBinding            =  hoistedOldUpdate( getBinding )( "database"
)
dummyDatabase                  :=  oldDbBinding( get )()

let ok      := oldUpdate( cut )()
ok          := oldUpdate( add )( newUpdate )
```

Figure 7: Replacing the *update* procedure using Octopus.

3 Propagating Type Changes

On its own, the Octopus mechanism is not suitable for evolving programs in which changes of type have occurred. This section describes an extension of the Octopus mechanism which allows type dependent information inside programs to be manipulated. The mechanism is an extension of Octopus in that type specific information is extracted from executable code and stored inside the environment vector.

When a type used by a program changes, it is not always necessary to make syntactic changes to the program in order to reflect the new type. For example, if the type of *Part* in the *partsDatabase* example is extended with a new field, the *update* operation does not need to be changed syntactically, but must be recompiled in order to update any type specific information in its executable form.

The *create* and *display* operations must be modified to include the new field. The nature of the resulting changes in executable code depend on how each procedure uses the type information.

In general, programs depend on types in three ways. These are classified as:

1. signature,
2. field, and
3. constructor.

The nature of these dependencies and their effects on the executable forms of programs are described below.

Programs which contain signature dependencies contain code to perform type checking on values dynamically. For example consider the *display* procedure in the parts database example. This procedure attempts to locate procedures called *writeString* and *writeInt* in the persistent store. Locating the procedures requires type checking to be performed dynamically. For this reason, representations of the types of the procedures are kept in the persistent store. In order to locate the procedures, the executable code of *display* contains a representation of the expected types of the procedures and uses them to compare with the representation encountered in the persistent store.

The executable form of a program containing a signature dependency is bound to a representation of the type. In the event that a change occurs in the signature, the source code may or may not require modification. However, the only change required in the executable form is that it should be bound to the new type representation.

Field dependencies arise in programs which dereference or assign to record fields. For example, the *update* procedure makes both a dereference and assignment to a *Part* field. The executable code contains information about the offset of the *quantity* field in the part. If, for example, a new field were added to *Part*, then the source code for *update* would remain the same: the only difference would be that the executable code would require updated information about the offset of the *quantity* field in the *Part* record. In such cases, recompilation could be avoided by updating the offset information bound to the executable code.

A constructor dependency arises when a program constructs an instance of a type. For example, in the case of the *create* procedure, the executable code constructs an instance of the type *Part* using the values passed as parameters to the procedure. If the type *Part* changes, the program *create* would also need to be changed. This involves syntactically changing the source code of *create* and recompiling it.

3.1 Operations for Evolving Types

The dependencies described above indicate that not all programs need to undergo recompilation if the types they use change. More specifically, if programs contain only signature and/or field dependencies, the programs may be evolved without the expense of recompilation if the types they use are evolved. The three operations in Figure 8 provide a way in which the type dependencies of programs may be queried, and the programs updated without recompilation where possible.

The *updateType* operation replaces representations of type *old* with representations of type *new* in the procedure *aProc*. If *aProc* contains field dependencies then the offsets are also updated. If the procedure contains occurrences of structurally equivalent types, then all such instances are updated; it is necessary to recompile the procedure if a subset of all equivalent types requires updating*. The *getDepend* operation returns a value indicating the nature of the dependency of *aProc* on the type *theType*. These values are integers combined from the following:

0 no dependency,

*In programming languages which employ name equivalence on types, this would not necessarily be the case.

```
                    1       signature dependency,
                    2       field dependency,
                    4       constructor dependency, and
                    8       semantic dependency of some other kind.

updateType  :  proc(   aProc  : any ; old,new : Type → bool )
getDepend   :  proc(   aProc  : any ; theType : Type → int )
scanTypes   :  proc(   aProc  : any ;
                       user   : proc(  aType    : Type ;
                                       aDep     : int ) )
```

Figure 8: Operations for type evolution and dependency querying.

Combinations are produced by adding the appropriate values; for example, the value 5 indicates a constructor and signature dependency. The final operation, *scanTypes*, allows a user specified procedure to be iteratively applied to each dependent type of the procedure *aProc*. The operations described above may be used to more efficiently evolve an application in which type changes occur. However, the operations do not provide a way of avoiding recompilation in all cases: if a program requires syntactic change, then recompilation is unavoidable.

4 Nodules

The operations provided to support type evolution behave in a similar fashion to the Octopus operations. In Octopus, all bindings are removed from the executable code stream and stored separately in an environment vector. This principle is extended to support type evolution by extracting all type representations and field offsets from the code stream and planting them in the environment vector with the bindings. For example, Figure 9 shows the environment vector of the *update* procedure in the parts database storing the offset of the *quantity* field inside a part.

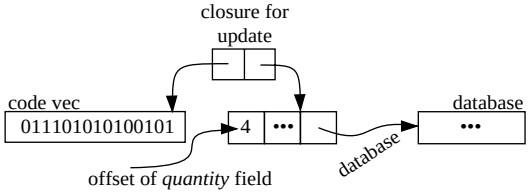

Figure 9: Storing type information in environment vectors.

The result is that all attributes of the executable code which may be modified without recompilation are stored separately. Consequently, the reusability of executable code is considerably extended: the same executable code may be used with a different environment vector to operate on different values with different types.

One method of constructing general purpose code is to use generators [4]. This technique involves parameterising a procedure with all relevant intermediate free variables. For example, the *update* procedure needs to be bound to a database, so a generator may be constructed which takes a database as a parameter and returns an update procedure bound to that database as shown in Figure 4.2.

```
let updateGen = proc( db : Database → proc( string,int ) )
begin
        proc( partName : string ; amount : int )
        begin
            let part = db( lookup )( partName )
            part( quantity ) := part( quantity ) + amount
        end
end
```

Figure 10: A generator for *update* procedures.

The generator in Figure 10 has a single parameter *db* of type *Database*. All the instances of *update* produced by *updateGen* may be potentially bound to different databases which have the same type, namely *Database*. This may be generalised by the construction of a generator which is parameterised by the type of the database, i.e. to use a form of polymorphism. However, it is not possible to construct an unbounded universally quantified polymorphic generator since there are implicit constraints on the type of the database. Each generated procedure requires that the database has a field called *lookup* which is a procedure that takes a string as a parameter and returns a record. Furthermore, the returned record must have an integer field called *quantity*. These constraints require bounded universal quantification.

Constructing bounded universally quantified procedures adds considerable complexity to the applications being constructed. Ideally, such complexity would be hidden by appropriate tools in the programming environment. A new mechanism called a *Nodule* (Napier module) [11] is introduced which provides this ability. Nodules provide a way in which generic code may be specified and reused. They may be highly parameterised by the values and types used by a program, thus forming a template from which specialised instances may be obtained. Each specialised instance of a Nodule is bound to the same generic executable code as illustrated in Figure 11.

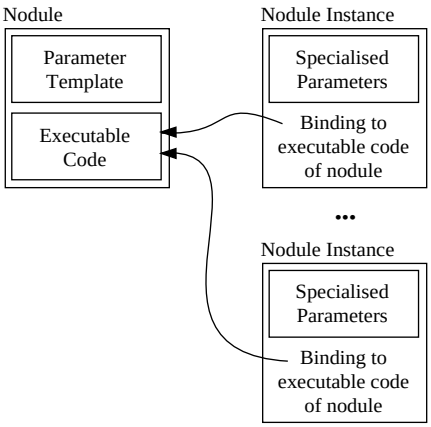

Figure 11: Conceptual view of Nodules and instances.

Nodules consist of five parts: type parameters, value parameters, functional parameters, executable code and an interface description of each Nodule instance. A Nodule is constructed and instantiated independently. Nodule

instantiation involves supplying some or all of the necessary parameters to the Nodule in order to produce either a complete specialised instance or a more restricted Nodule. The latter technique is akin to *currying* and results in a new Nodule with fewer unbound parameters.

The Nodule in Figure 12 describes a parameterised version of the *update* procedure used in constructing the *partsDatabase* application.

```
NODULE UpdateNodule
PARAMETERS
    TYPE Database
    VALUE database: Database
INTERFACE
    update: proc( string,int )
```

```
SOURCE
let update= proc( name : string ; amount : int )
begin
    let part= database( lookup )( name )
    part( quantity ) := part( quantity ) + amount
end
```

Figure 12: A Nodule describing a parameterised *update* procedure.

The type *Database* and the database instance are parameters to the Nodule. There are no explicit constraints upon the type of the database; however, implicit constraints do exist: firstly, the Nodule code constrains the type *Database* to be a type which has the appropriate functionality. In this context, the type *Database* must contain the function *lookup*, which takes as its parameter a string, and returns a record which must contain an integer field, *quantity*. When a concrete type *Database* is supplied to the Nodule along with a database, a fully bound executable procedure of type **proc**(**string**,**int**) is obtained. The generic code contained in a Nodule represents a form of bounded universal quantification. However, this polymorphism is not provided by the programming language, Nodules are a mechanism used within the programming environment.

5 Environment Support

Nodules are not a feature of the programming language, instead they reside in the programming environment. In addition to tools which provide a graphical user interface to Nodules, the programming environment contains tools which allow the persistent store to be navigated, and for types and values to be located. Using a combination of these tools, users may locate Nodules, use a browser to discover values and types, and instantiate Nodules using the discovered entities. Thus, applications may be constructed without the need to write programs.

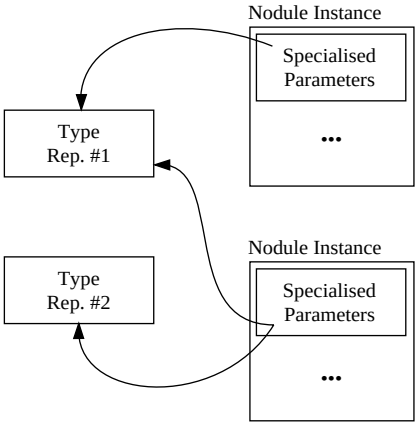

Figure 13: Storing type dependencies in Nodule instances.

Once Nodule instances have been created, they may be used as parts of other applications. The instances are values in the programming language space and therefore, if necessary, they may be manipulated by Octopus and the type evolution operations. Furthermore, Nodule instances contain information about the types on which they depend. By querying an application using Octopus, the Nodule instances dependent on a particular type may be discovered in addition to the types on which any particular Nodule instance depends. Therefore, given a particular application constructed using Nodules, if a change occurs in the schema of the application then all affected nodes may be located. Using the evolutionary mechanisms described earlier, the necessary components may then be evolved. This arrangement is shown in Figure 13.

6 Conclusions

We have presented a number of mechanisms which provide support for evolution and reuse in a persistent programming environment. Octopus permits systems of code and data to be evolved, even if they are encapsulated. This is achieved by separating pure code from bound values and providing meta level operations on all values, regardless of their type. The difficult problem of type evolution is addressed by extending the Octopus architecture to separate type information from the executable code. In many cases type evolution is possible, without the expense of total or partial system recompilation.

Nodules, a programming environment mechanism for specifying reusable components, has also been described. Nodules are complementary to the mechanisms described above and allow parameterised templates to be defined independently of any referencing environment. Nodules may be specialised in order to yield components by binding them to values and types. Since this is supported by the environment, the activity of application construction may be performed without the need to write programs. Nodules also provide added support for evolution by recording the relationship between the schema and bound Nodule instances.

Acknowledgements

We would like to thank John Rosenberg, Karen Wyrwas, Sam Bushell and David Hulse for their comments on this paper. This work was supported by the Defence Science and Technology Organisation of Australia.

References

1. Booch, G. "Object Oriented Design", Benjamin-Cummings, 1991.
2. Connor, R. C. H. "The Napier Type-Checking Module", Universities of Glasgow and St Andrews, Technical Report PPRR-58-88, 1988.
3. Dearle, A. and Brown, A. L. "Safe Browsing in a Strongly Typed Persistent Environment", *The Computer Journal*, vol 31, 6, pp. 540-545, 1988.
4. Dearle, A., Cutts, Q. and Connor, R. "Using Persistence to Support Incremental System Construction", *Microprocessors and Microsystems*, vol 17, 3, pp. 161-171, 1993.
5. Farkas, A. and Dearle, A. "The Octopus Model and its Implementation", in *Proceedings of the 17th Australian Computer Science Conference, Australian Computer Science Communications*, vol 16, pp. 581-590, 1994.
6. Farkas, A. and Dearle, A. "Octopus: A Reflective Language Mechanism for Object Manipulation", in *Proceedings of the Fourth International Workshop on Database Programming Languages*, New-York, Springer-Verlag, 1994.
7. Kirby, G. N. C., Connor, R. C. H., Cutts, Q. I., Dearle, A., Farkas, A. M. and Morrison, R. "Persistent Hyper-Programs", *5th International Workshop on Persistent Object Systems*, San Miniato, *Persistent Object Systems*, Springer-Verlag, Workshops in Computing, pp. 86-106, 1992.
8. Liskov, B. H. and Zilles, S. N. "Programming with Abstract Data Types", *SIGPLAN Notices*, vol 9, 4, 1974.
9. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. "Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment", Universities of Glasgow and St Andrews, Technical Report PPRR-32-87, 1987.
10. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. "The Napier88 Reference Manual", University of St Andrews, Technical Report PPRR-77-89, 1989.
11. Farkas, A. and Dearle, A. "Integrated Support for Incremental Software Development and Evolution", University of Adelaide, Technical Report PS-24, May 1994.