Evolution and Incremental Construction Support via Integrated Programming Environment Mechanisms

Alex Farkas and Alan Dearle

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

{alex,al}@cs.stir.ac.uk

Abstract

The mechanisms described in this paper support a software engineering environment in which the transition from initial design through implementation and into maintenance is a smooth and continuous process. Two new datatypes, Nodule and Octopus, are presented. Nodules are templates containing compiled code and labelled typed locations, and are intended to support the interactive construction and evolution of applications. The major benefit of the Nodules over other systems is that they permit application systems to be generated that are complete, internally consistent and strongly type checked. The Octopus mechanism permits executable application systems to be evolved in situ. Using this mechanism, the information that was available to the application developer is accessible to the maintenance programmer. It also enables components of the application to be evolved or replaced, and reconnected to live data in a type safe manner. When combined into a single system, the Nodule and Octopus datatypes 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

This paper presents a model to support application systems during their design, implementation and maintenance. The goal of the model is to allow application systems to be seamlessly evolved from prototypes into production level systems via direct interaction with the user [4]. This is achieved via the provision of parameterised templates containing compiled code.

The mechanisms described in this paper support a software engineering environment in which the transition from initial design through implementation and into maintenance is a smooth and continuous process. As in conventional software engineering environments, the software engineer designs an application by defining the major components of the application and their interfaces, and then repeatedly refining each component by defining their sub-components. However, at any point during the design process, actual code may be associated with any of the components, and testing performed on those completed parts of the application. In this way, the design and implementation may be tightly coupled [15]. Components can be added or removed, their interface or behaviour changed, affecting both the design and implementation simultaneously. The software engineering system would check the application under construction for consistency, and provide feedback to the programmer. When the design has been completed and code associated with all the components, the final application is produced which is guaranteed to be internally consistent. Later, if the finished application needs correcting or evolving, a maintenance programmer, or an update program, could break the application open to operate over the same components used by its designers and implementors, information normally encapsulated within the application program and therefore inaccessible. This enables components of the application to be evolved or replaced, and reconnected to live data.

The paper is structured as follows: Section 2 describes an abstract model for program development. Section 3 specifies in detail an implementation of this model, Section 4 demonstrates its application. Section 5 describes a mechanism for evolving and maintaining executable applications, Section 6 shows how this may be applied.

2 The Abstract Model

The model described here is an extension of the architecture described in [7] in which applications are constructed that are internally bound using L-value (location) bindings. The programmer is provided with a method of accessing these locations, enabling the contents of the locations to be changed, thereby changing the behaviour of the system as a whole. No dynamic binding or type checking is required when an application is executed since all binding is performed at the time analogous to link time in traditional programming systems. This architecture has been used to successfully implement a number of applications including compilers and database applications. However, these applications were constructed without any environment support and relied on the application programmers to maintain the locations manually and to follow programming conventions when establishing bindings.

The primary abstraction used in the model is that of a template; templates contain compiled code and may be thought of as prototypes for values[†]. In this respect, templates are similar to classes found in object oriented languages. Templates contain a distinguished piece of code, and labelled typed locations. The code provides the interface to instances created from the template, locations may contain either values, templates, or bindings to other locations. A template is shown in Figure 1, it contains three typed locations *A*, *B* and *C*. Locations *A* and *B* contain templates, while location *C* contains a value which is a procedure. The template stored in location *A* contains two locations: *X* is uninitialised and *Y* contains a value with a binding to location *C*. The template stored in location *B* also contains two locations: location *P* contains a value that contains a binding to location *A*, *Q* contains a value with a binding to location *P*.

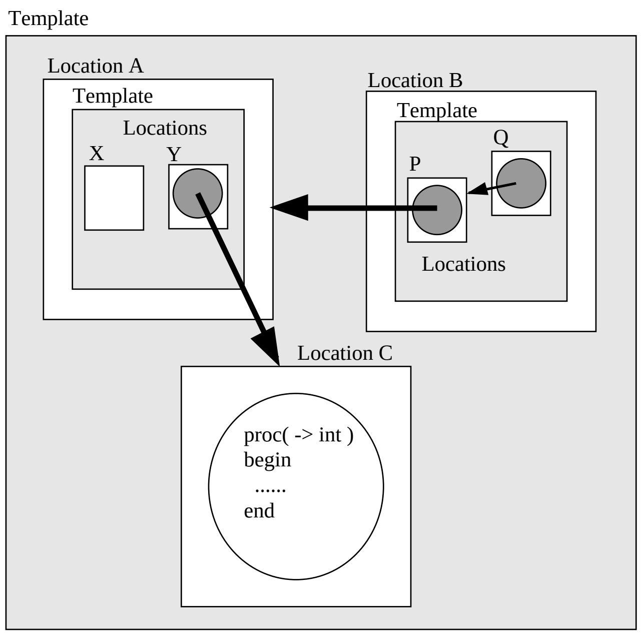


Figure 1: An example template.

The distinguished code contained in a template may make bindings to a location in any of the templates in which it is resident; for example, the code in *Y* may bind to the locations *X*, *B* and *C*. All bindings are L-value bindings; that is bindings are made to locations and not the data stored in those locations. Using an assignment operator, the data stored in a location may be updated with any template or value that is type compatible with the type of the location. Consequently, the behaviour of any code that is bound to an updated location will automatically

[†] The term *value* as used in this paper is used to mean any values from the programming language value space. This may include arbitrary scalars, constructed data structures, procedures, functions, ADTs and objects.

reflect the changes. However, sometimes a modification will require that the type of a component be changed; such a type change may necessitate a change in type of a location which is bound to other templates and values. For example, the location denoted by *C* in Figure 1 is bound to the value stored in the location denoted by *Y*. Clearly, some mechanism must be provided to support the change in the types of bound values and templates.

Consider the binding graph shown in Figure 2 in which all the values are bound directly or indirectly to the black location in the centre of the diagram. Should the type of this location change, the values or templates stored in the locations shaded in grey must be rebound since any change to the type of the black location will render their specification incorrect. Therefore, when a type change occurs to a location, any bindings to that location are automatically dissolved. The values or templates stored in the grey locations must be rebound, perhaps after modification. The values and templates stored in the white locations bind to the grey locations rather than the values stored in them. Consequently, they will be unaffected by any changes to the black location.

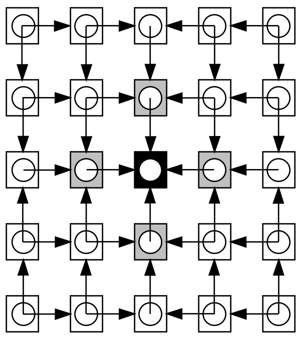


Figure 2: A template binding graph.

Unlike modules in most programming languages [9, 10, 12], templates are not defined in a single syntactic entity. Instead, templates are created containing nothing, all they possess is their identity, and are incrementally evolved to the desired level of functionality. This is achieved by adding typed named locations and (perhaps incomplete) code, by establishing bindings between code and locations, and by assigning values and (perhaps incomplete) templates to locations. Thus the incomplete template shown in Figure 3a may be evolved into the template shown in Figure 3b.

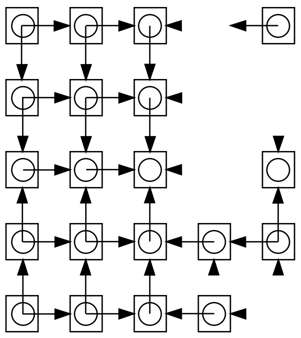


Figure 3a: A template under construction.

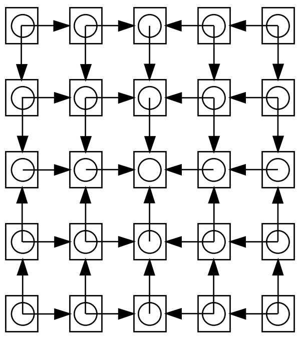


Figure 3b: A complete template.

In order to support the incremental construction of applications, bindings to locations need not be totally resolved when the code is supplied to the template. However, the name and type of any unresolved bindings must be specified in order to allow the code to be checked for correctness. For example, consider the code for a *push* stack operation shown in Figure 4 which makes use of two free variables *pos* and *stack*. The *push* procedure may be described in a template with the code bound to two locations *pos* and *stack* with appropriate types. These locations could be bound later to other locations that are perhaps also bound to templates for other stack operations such as *pop* and *top*. Alternatively, the locations could be initialised with values to allow the *push* template to be unit tested in isolation of any context. Following this option would not preclude the use of the former when unit testing was complete.

```
!** pos     : int
!** stack   : array of int

let push = proc( i : int )
begin
        pos := pos + 1
        stack( pos ) := i
end
```

Figure 4: The push stack operation.

Throughout the software life-cycle, the engineers working on an application must be aware of the components from which the application is constructed, and how those components are interconnected. Therefore, the template mechanism provides a set of query functions over the templates that allow the template structure to be discovered.

Application programs consist of a distinguished piece of code bound to a directed graph whose nodes are locations containing arbitrary values. In the model described here, an application is produced by instantiating a template. This constructs a graph that is isomorphic to the template binding graph, but with each location containing a value wherever a template exists in the original graph. In order to instantiate a template, all the bindings contained in the template must be resolved, bindings to uninitialised locations are not permitted.

The stack example described earlier highlights another requirement of the model, namely the need for some locations to be dynamically instantiated in the same manner that local variables are constructed in procedures of algol-like languages. For example, every instance of a stack package should have its own copy of the variables *pos* and *stack*. To facilitate this, the model allows some locations to be specified to be *local* to a template. Local locations are created dynamically when the code associated with the template is executed, whereas ordinary locations are created when the application is instantiated.

After an application is generated, it may still be subjected to evolutionary pressures. For example, consider a banking system that has been constructed in an object oriented manner. Should the account data be encapsulated within a faulty object, it would be desirable to have the ability to correct the bug and evolve the component without losing the encapsulated bank accounts. Furthermore, the programmer maintaining the system should be allowed access to the data that was available during the construction of the component. In the model described in this paper, the internal structure of an arbitrary value, including an application program, may be discovered by converting the value into an Octopus [8]. Normally this information is invisible and encapsulated within the application program. When viewed as an Octopus, components of the application may be evolved or replaced and reconnected to live data.

The Octopus mechanism provides a dynamic infinite union type with a set of reflective operations. 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.

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, values may be *dropped* back into the value space, provided that they still conform to the language's type system.

The Octopus model provides the ability to *cut* and *rewire* the bindings within a hoisted value in much the same way as templates support this activity during the construction phase. 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 neither possible to rewire values of an incompatible type nor drop hoisted values back into the value space whilst cut bindings exist.

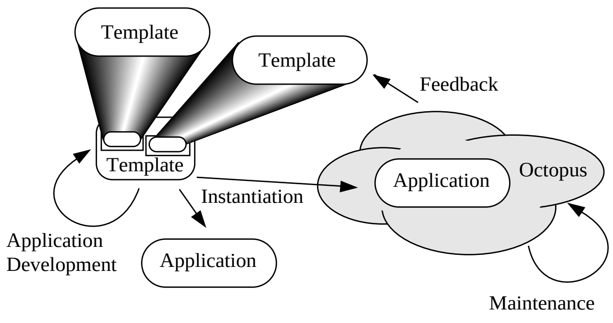


Figure 5: The application lifecycle.

The lifecycle of an application is shown in Figure 5. A newly created template is merely an empty shell, it is enhanced by adding code, locations, other templates and values to it. An application may be created by instantiating the template. If the application later requires maintenance, it may be injected into an Octopus which provides a view of the application's internal state. When viewed as an Octopus, new values, perhaps generated from other templates, may be assigned to the locations contained in the application and bound to its internal state. Changes made to the instance may be propagated back to the original template from which the instance was produced in order that future application instances also contain the correction. When the application has been modified, it may be dropped back into the value space of the programming language.

3 The Nodule Datatype

One instance of the above method of program construction is described for the persistent programming language Napier88 [14]. In this architecture, a program template is realised by an abstract data type called a *Nodule* (Napier module). The Nodule operations allow an application template to be developed which may later be instantiated to produce an executable application. Nodules contain three abstractions: locations, locals and source code; these correspond to the locations, locals and distinguished code described above.

Each Nodule contains a set of typed locations which may be added incrementally and are used for storing values, Nodules and bindings to other locations. The locations are independent of the data stored in them and may have data assigned to them at any time prior to Nodule instantiation. Nodules also contain a set of typed *locals*; these are used to denote values which are dynamically instantiated whenever an instance produced from a Nodule is executed. The final component of a Nodule is its source code which may contain references to locations or locals within the Nodule.

```
type CompResult is variant( Ok : null ; Fail : List[ string ] )

type SourceResult is variant(     Valid,Invalid : string ; Unspecified : null )

type NameKind is variant( Location,Local : string )

type BindingResult is structure( SrcName,Destname : NameKind )

type CellInfo is structure(  Name : string ; Type : TypeRep )

rec type CellResult is variant(    Value : any ; Nodule : Nodule ; Unbound : null )

&     Nodule is structure(
              mkInterface:        proc( Type : TypeRep );
              getInterface:       proc( → TypeRep );
              mkSource:           proc( source : string → CompResult );
              getSource:          proc( → SourceResult );
              mkLocal:            proc( key : string ; Type : TypeRep → bool );
              deleteLocal:        proc( key : string → bool );
              getLocalInfo:       proc( → List[ CellInfo ] );
              getLocal:           proc( key : string → CellResult );
              mkLocation:         proc( key : string ; Type : TypeRep → bool );
              deleteLocation:     proc( key : string → bool );
              getLocationInfo:    proc( → List[ CellInfo ] );
              getLocation:        proc( key : string → CellResult );
              assignNoduleToLocal:        proc( key : string ; N : Nodule → bool );
              assignValueToLocal:         proc( key : string ; a : any → bool );
              assignNoduleToLocation:     proc( key : string ; N : Nodule → bool );
              assignValueToLocation:      proc( key : string ; a : any → bool );
              bind:               proc( sourceName, targetName : string → bool );
              getBindingInfo:     proc( → List[ BindingResult ] );
              newInstance:        proc( →  any ) )

!** Nodule constructor function
newNodule       :       proc( → Nodule )
```

Figure 6: Type declarations and signatures describing Nodules.

The interface to a Nodule is shown in Figure 6. The type *TypeRep* which is not declared in Figure 6, is a structured representation of types used by the Napier88 system. The Napier88 system provides a complete set of selector, constructor, equivalence and iterator functions that operate over type representations [5]. For simplicity we assume that the types package includes a function called *mkTypeRep* which constructs a type representation from a string‡. The type *any* is an infinite union type; values of any type may be injected into the type *any* and all consequent type checking over the injected values is performed dynamically. The semantics of the Nodule operations are described below.

‡ In practice, the types module contains lower level functions than this. However, such a function may be constructed using the types module.

mkInterface	defines the type of instances produced from a Nodule by calling the *newInstance* function. If a different interface already exists, and source code has been specified, then the source code is invalidated.
getInterface	returns a representation of the type of the instances produced from a Nodule by calling the *newInstance* function.
mkSource	compiles the specified source text in the context of the locations and locals defined in the Nodule. The operation fails if the type of the source is incompatible with the interface of the Nodule, or if an interface has not been specified, or if the compilation fails.
getSource	returns a copy of the source code (if any) of the Nodule and the state of this source code.
mkLocal	defines a dynamically instantiated local. This operation fails if a local or location of the same name already exists in the Nodule.
deleteLocal	removes a dynamically instantiated local and invalidates any code, locations or other locals bound to the local. This operation fails if a local with name *key* does not exist.
getLocalInfo	returns a list of the names and types of each local in the Nodule.
getLocal	returns the state of a local with name *key*. If a value or Nodule is stored in this local the data is also returned.
mkLocation	defines a typed, empty location which may contain a value, a Nodule, or a binding to another location. This operation fails if an item of name *key* exists in the Nodule prior to the call.
deleteLocation	removes the location with name *key* and invalidates any bindings to that location. This operation fails if a location with name *key* does not exist.
getLocationInfo	returns a list of the names and types of each location in the Nodule.
getLocation	returns the state of a location with name *key*. If a value or Nodule is stored in this location the data is also returned.
assignNoduleToLocal	assigns a copy of the Nodule *N* to the local called *key*. This operation fails if the type of *N* does not match the type of the local, or if a local with name *key* does not exist in the Nodule.
assignValueToLocal	assigns the value injected into *a* to the local called *key*. This operation fails if the type of *N* does not match the type of the local, or if a local with name *key* does not exist in the Nodule.
assignNoduleToLocation	assigns a copy of the Nodule *N* to the location called *key*. This operation fails if the type of *N* does not match the type of the location, or if a location with name *key* does not exist in the Nodule.
assignValueToLocation	assigns the value injected into *a* to the location called *key*. This operation fails if the type of *N* does not match the type of the location, or if a location with name *key* does not exist in the Nodule.
bind	Binds the location or local called *sourceName* to the location or local called *targetName*. The name sourceName may be a structured name with dots

indicating context. For example, in Figure 1, the location called *Y* may be bound to location *C* using the operation *bind(* "A.Y","C" *)*. This operation fails if any of the names do not exist, or if the abstractions have different types. Locations may not be bound to locals within the same Nodule; consequently, if this is attempted the operation will also fail.

getBindingInfo
returns a list of all bindings in the Nodule made using the *bind* operation.

newInstance generates a value using the Nodule as a template. The type of the value generated is that specified by *mkInterface*. The operation fails if the Nodule is internally inconsistent or incomplete.

newNodule returns a new empty Nodule.

4 An Example Application using Nodules

```
let stackGen = proc( → structure(   push  :    proc( int )
                                    pop   :    proc()
                                    top   :    proc( → int )
begin
    let stack = array 1 to 100 of 0     !** Create a zero filled array of size 100.
    let pos := 0                        !** Declare an integer to use as the stack index.

    let push = proc( i : int )!** Increment pos and assign value to new stack offset.
    begin
        pos := pos + 1
        stack( pos ) := i
    end

    let pop = proc()                    !** Decrement the stack index.
    begin
        pos := pos – 1
    end

    let top = proc( → int )             !** Return the item at the top of the stack
    begin                               !** the stack index.
        stack( pos )
    end

    struct(   push =    push ;!** Construct package of stack operations.
              pop  =    pop ;
              top  =    top )
end
```

Figure 7: A program to generate packages of stack operations.

To illustrate how an application is constructed using Nodules, consider the Napier88 stack package shown in Figure 7; this example has been chosen for simplicity and brevity. Nodules are intended for much larger and higher level applications than the stack example since the benefits of the mechanism are reduced as the complexity of the application decreases. In the case of the stack package, the structure of the application is readily apparent through inspection of the entire package. In a larger application, querying tools are required to allow the programmer to discover the structure of the application under development [2, 3]. In the case of the stack package, any change to a component could easily be achieved by a single programmer modifying and recompiling the entire package. Furthermore, the entire package is so small that unit testing of individual components in isolation makes little sense.

The procedure *stackGen* in Figure 7 returns a package of operations bound to a stack implemented as an array of integers. As is common in Napier88 programs, the package is implemented as a record containing procedures each bound to encapsulated values [1, 6]. New encapsulated values (*stack* and *pos*) and new procedure closures (*push*, *pop* and *top*) are created upon each invocation of the generator.

```
let stackGenN = newNodule()

stackGenN( mkInterface )( mkTypeRep( "proc( → structure(  push  : proc( int );
                                                          pop   : proc();
                                                          top   : proc( → int ) )" ) )

stackGenN( mkLocal )( "localpos",      mkTypeRep( "int" ) ) )
stackGenN( mkLocal )( "localstack",    mkTypeRep( "array of int" ) ) )
stackGenN( mkLocal )( "push",          mkTypeRep( "proc( int )" ) )
stackGenN( mkLocal )( "pop",           mkTypeRep( "proc()" ) )
stackGenN( mkLocal )( "top",           mkTypeRep( "proc( → int )" ) )

stackGenN( mkSource )( "proc( → structure(  push  : proc( int ) ;
                                            pop   : proc() ;
                                            top   : proc( → int ) )
                        begin
                              struct( push = push ; pop = pop ; top = top )
                        end" )
```

Figure 8: The history of operations on the *stackGenN* Nodule.

A log of a typical interaction to construct the *stackGen* application is shown in Figure 8. Code such as this would generally be executed in response to the interaction of the application designer with an environment tool. The application designer could choose to construct the application in a different order to yield the same result. First, a Nodule representing the *stackGen* application must be created using the *newNodule* operation. Initially, the Nodule is empty: it does not contain any locals, locations, or source code, nor does it have a specific interface. Later the implementation of the stack package is developed by calls to the Nodule constructor functions, in this case, *mkInterface*, *mkLocal* and *mkSource*. The type of the values generated from the Nodule is specified using *mkInterface*. Next, five locals are added by specifying their name and type in calls to *mkLocal*. Finally the source code for the package is specified by a call to mkSource; this source code binds to the, as yet uninitialised, locals *push*, *pop* and *top*.

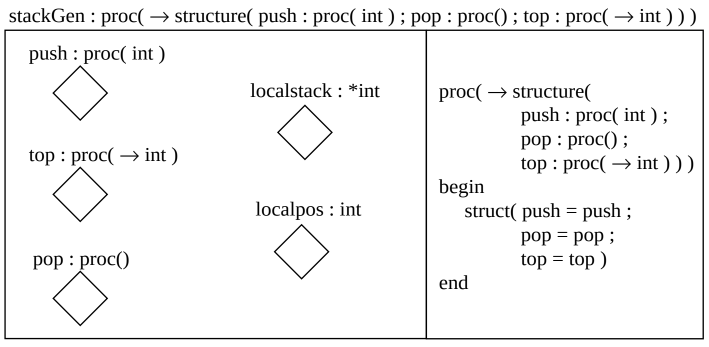


Figure 9: A view of *stackGenN* after the first stage of construction.

A view of the *stackGenN* Nodule after the sequence of steps shown in Figure 8 have been executed is shown in Figure 9. The source code is shown on the right side of the Nodule and the locals are on the left. The locals are shown as empty diamonds indicating that they have not yet been initialised. The interface to the Nodule appears alongside the Nodule name at the top of the figure.

The next step in constructing the stack generator is to create Nodules for each of *push*, *pop* and *top*. These three Nodules may be constructed in any order and independently of each other. Alternatively, predefined templates may already exist in the programming environment which may be reused. Figures 10, 11 and 12 show the history of the calls necessary to construct these Nodules.

```
let pushN = newNodule()
pushN( mkInterface )( mkTypeRep( "proc( int )" ) )
pushN( mkLocation )( "stack",      mkTypeRep( "array of int" ) )
pushN( mkLocation )( "pos",        mkTypeRep( "int" ) )
pushN( mkSource )( "  proc( i : int )
                      begin
                            pos := pos + 1
                            stack( pos ) := i
                      end" )
```

Figure 10: The *pushN* Nodule.

```
let popN = newNodule()
popN( mkInterface )( mkTypeRep( "proc()" ) )
popN( mkLocation )( "pos",mkTypeRep( "int" ) )
popN( mkSource )( "   proc()
                      begin
                            pos := pos – 1
                      end" )
```

Figure 11: The *popN* Nodule.

```
let topN = newNodule()
topN( mkInterface )( mkTypeRep( "proc( → int )" ) )
topN( mkLocation )( "stack",mkTypeRep( "array of int" ) )
topN( mkLocation )( "pos",mkTypeRep( "int" ) )
topN( mkSource )( "proc( → int ) ; stack( pos )" )
```

Figure 12: The *topN* Nodule.

The three Nodules are similar to each other; we shall therefore only discuss the construction of the *pushN* Nodule. *PushN* contains two locations, *stack* and *pos*, to which the source code of the Nodule binds. These are specified to be locations rather than locals since they represent unbound intermediate free variables which will be later bound to the *stackGenN* context. The complete *pushN* Nodule is shown in Figure 13 in which the *stack* and *pos* locations are shown as empty, plain boxes indicating that they are uninitialised.

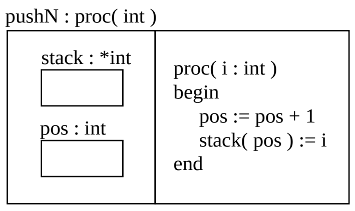
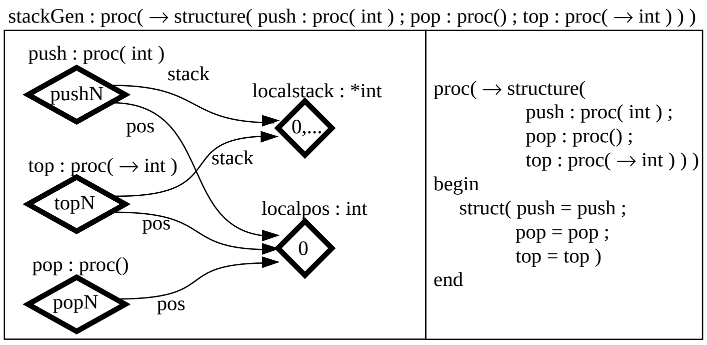


Figure 13: The *pushN* Nodule.

At this point, all of the necessary Nodules have been constructed and may be composed to form a complete application template as shown in Figure 14. The first two lines specify the initial values for the locals *localpos* and *localstack*. The next three lines assign copies of the respective Nodules *pushN*, *popN* and *topN* to the corresponding locals in the *stackGenN* Nodule. Since these are locals, the stack generator will construct new procedures from the Nodules each time it is invoked. This is the correct semantics since these procedures should bind to local copies of *localpos* and *localstack*. Procedures with no bindings to free variables may safely be assigned to locations rather than locals. The final five lines establish bindings between the Nodule components. For example, the first call to bind indicates that the location denoted by *stack* within the *pushN* Nodule should be bound to *localstack* in *stackGenN*. The completed application template is shown in Figure 15.

```
stackGenN( assignValueToLocal )( "localpos",        any( 0 ) )
stackGenN( assignValueToLocal )( "localstack",      any( array 1 to 100 of 0 ) )

stackGenN( assignNoduleToLocal )( "push",   pushN )
stackGenN( assignNoduleToLocal )( "pop",    popN )
stackGenN( assignNoduleToLocal )( "top",    topN )

stackGenN( bind )( "push.stack",   "localstack" )
stackGenN( bind )( "push.pos",     "localpos" )

stackGenN( bind )( "pop.stack",    "localstack" )

stackGenN( bind )( "top.stack",    "localstack" )
stackGenN( bind )( "top.pos",      "localpos" )
```

Figure 14: Composing the Nodules to form a complete stack Nodule.



```
proc( → structure(
            push : proc( int ) ;
            pop : proc() ;
            top : proc( → int ) ) )
begin
    struct( push = push ;
            pop = pop ;
            top = top )
end
```

Figure 15: The complete stack Nodule.

Once the stack Nodule is complete, a new application program may be produced by calling the *newInstance* operation from *stackGenN*. Provided that the Nodule is complete this will return the application program injected into *any*. The value may be obtained as shown in Figure 16 and applied to produce an instance of a stack package.

```
type StackPackGen is proc( → structure( push : proc( int ) ; pop : proc() ; top :
proc( → int ) )

let anystackGen = stackGenN( newInstance )()
let myStack =     project anystackGen as stackGen onto
                                  StackPackGen      : stackGen()
                            default                 : error( ... )

myStack( push )( 7 )
```

Figure 16: An operational Stack package.

5 Evolution using Octopus

During their lifetime, it is inevitable that applications undergo change, this may be to correct errors or simply to improve their functionality. It is desirable to make such changes with the minimum of interference to a working application. Furthermore, as discussed earlier, it is essential to maintain the state of the system prior to evolution. The Octopus mechanism provides this functionality by enabling the internal structure of an application program to be interrogated and manipulated. Two special operations exist which allow a value to be hoisted into and dropped from an Octopus. Type declarations for the two operations are shown in Figure 17.

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

Figure 17: The Octopus hoist and drop operations.

The first operation, *coerceToOctopus*, hoists a value into an Octopus. 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.

An Octopus comprises a set of four operations which allow the bindings within a value to be examined and manipulated. These operations are implemented as a package of functions as shown in Figure 18; the hoisted value is encapsulated in the closure of these functions.

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

Figure 18: 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 *getSource* operation returns a copy of the Nodule used to construct the value. 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. Bindings are also represented as a package of functions, and are described below.

5.1 Bindings

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

Figure 19: The representation of a binding.

Each binding is represented by six operations, as shown by the corresponding type declaration in Figure 19. 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.

Using the operations described in this section, an application graph may be examined in order to locate and extract components for replacement or reuse. In particular, the state of an old component may be obtained and wired into a replacement component in order to maintain the state of an application. This process is illustrated in the next section.

6 Evolving the Stack Example using Octopus

This section demonstrates how an instance of the stack package may be evolved using Octopus. Like the earlier example, this is kept simple for clarity of explanation. Octopus is intended to provide a platform for the construction of higher level maintenance tools rather than to be used in the manner demonstrated here.

```
let pos := 7                          !** any old value to allow the declaration of newPop.

let newPop = proc( → int )
begin
      if pos > 0 then
            pos := pos – 1
      else
            error( ... )
end
```

Figure 20: A correct *pop* procedure.

The more observant reader will have noticed that the *pop* operation in Figure 7 is erroneous – it does not check to see if the stack is empty before decrementing the stack index. To correct this problem, we shall construct the new *pop* procedure shown in Figure 20 and wire it into an instance of the stack package whilst retaining the data held in the stack; this is illustrated in Figure 21.

The *updatePop* procedure shown in Figure 21 updates the *pop* component of an arbitrary stack package as follows. Firstly, the *pop* procedure from the stack package and the *newPop* procedure are hoisted into Octopuses using the *coerceToOctopus* function. Next, the *pos* bindings from the old and new *pop* procedures are retrieved. The *pos* binding from the *pop* procedure is then cut and replaced with the binding to *pos* from the stack package. Once the *pos* value from the stack package is bound into the *newPop* Octopus, the encapsulated procedure is dropped back into the value space injected in an *any*. After projecting this any onto the correct type, the dropped value is assigned to the *pop* field of the supplied stack package completing the update. This final assignment is possible due to the fact that Napier88 supports first class procedures. In languages in which direct assignment is not possible, such as true object oriented languages, Octopus operations may be performed on the entire object, in this case the stack package, rather than on the interface procedure *pop*, to achieve the same result.

```
type StackPack is structure( push : proc( int ) ; pop : proc() ; top : proc( → int ) )

!** This procedure replaces the pop component of a stack package
let updatePop = proc( sp : StackPack ; newPop : proc() )
begin
        !** inject old pop into an Octopus
        let oldPopOctopus = coerceToOctopus( sp( pop ) )

        !** inject new pop into an Octopus
        let newPopOctopus = coerceToOctopus( newPop )

        !** get the pos binding from sp
        let spPos = oldPopOctopus( getBinding )( "pos" )

        !** get the pos binding from newPop
        let newPopPos = newPopOctopus( getBinding )( "pos" )

        let ok := newPopPos( cut )()              !** cut pos from  newPop
        ok := newPopPos( add )( spPos )           !** bind spPos for newPop

        let fixedPop = coerceFromOctopus( newPopOctopus )!** the new pop proc. in any
        project fixedPop as projFixedPop onto                ! ** project from the any
                proc()          :       sp( pop ) := projFixedPop      !** drop was successful
        default                 :       error( ... )                   !** the drop failed
end
```

Figure 21: Upgrading a stack package.

7 Conclusions

We have presented a programming model that supports the design, implementation and maintenance of application systems. A realisation of this model is provided by the Nodule and Octopus datatypes. Nodules are used to construct and evolve application templates, whilst Octopus enables evolution and maintenance to take place on executable applications.

In this paper, the Nodule and Octopus mechanisms are presented in their form as implemented in Napier88. However, it is envisaged that the principles behind the mechanisms could be applied to other programming paradigms, such as object-oriented or functional programming languages. For example, implementation of Octopus in an object-oriented system could be achieved by extending existing classes used by a system with the operations required to hoist, cut, rewire etc. Indeed, this tailoring could be applied to arbitrary depth in order to extend or restrict the available operations as required.

Nodules permit application systems to be developed interactively and incrementally with the same degree of freedom provided by systems such as Lisp [13] and Smalltalk-80 [11]. Like Lisp and Smalltalk, Nodules allow components to be tested in isolation. However, the major benefit of the model is that it allows application systems to be generated that are complete, internally consistent and strongly type checked.

The Octopus mechanism permits executable application systems to be evolved in situ. Such evolution may be achieved in interpreted, dynamically bound and type checked systems such as Lisp and Smalltalk. However, Octopus offers a number of advantages over such systems. Firstly, the Octopus mechanism is type safe; even though it allows encapsulated values to be discovered and manipulated, the integrity of the encapsulated data, and consequently the entire application, is never compromised. Secondly, the information that was available to the application developer is accessible to the maintenance programmer; the maintenance programmer never has to navigate untyped data structures such as Lisp *cons* cells and guess their meaning. Lastly, the Octopus mechanism may manipulate values containing compiled rather than interpreted code since the abstractions provided are at a high level.

When combined into a single system, the Nodule and Octopus datatypes 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.

Acknowledgements

This paper benefits from discussions with Karen Wyrwas.

References

1. Atkinson, M. and Morrison, R. "Procedures as Persistent Data Objects", *ACM TOPLAS*, vol 7, 4, pp. 539-559, 1985.

2. Bachman, C. W. "The Programmer as Navigator", Turing Award Lecture, Addison-Wesley, ACM Turing Award Lectures: The First Twenty Years 1966-1985, pp. 269-280, 1973.

3. Bachman, C. W. "The Programmer as Navigator, Architect, Communicator, Modeler, Collaborator and Supervisor", Postscript to 1973 Turing Award Lecture, Addison-Wesley, ACM Turing Award Lectures: The First Twenty Years 1966-1985, pp. 281-285, 1986.

4. Balzer, R. M. "Living in the Next Generation Operating System", *IFIP'86*, pp. 283-291, 1986.

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

6. Connor, R. C. H., Brown, A. B., Cutts, Q. I., Dearle, A., Morrison, R. and Rosenberg, J. "Type Equivalence Checking in Persistent Object Systems", *Implementing Persistent Object Bases*, Morgan Kaufmann, pp. 151 - 164, 1990.

7. Dearle, A., Cutts, Q. and Connor, R. "Using Persistence to Support Incremental System Construction", *Microprocessors and Microsystems*, vol 17, 3, pp. 161-171, 1993.

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

9. Futatsugi, K., Goguen, J., Meseguer, J. and Okada, K. "Parameterized Programming in OBJ2", *Proceedings of the Ninth International Conference of Software Engineering*, pp. 51-60, 1987.

10. Glassman, L. and Nelson, G. "Modula-3 Report", Technical Report 52 DEC SRC, Palo Alto, November, 1989.

11. Goldberg, A. and Robson, D. "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading, Massachusetts, 1983.

12. Ichbiah, J. D. "The Programming Language Ada Reference Manual", vol 155, Springer-Verlag, Lecture Notes in Computer Science, 1983.

13. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. "The Lisp Programmers' Manual", Technical Report, M.I.T. Press, Cambridge, Massachusetts, 1962.

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

15. Swartout, W. and Balzer, R. "On the Innevitable Intertwining of Specification and Implementation", *CACM*, vol 25, 7, pp. 438-449, 1982.