Using Persistence to Support Incremental System Construction

Alan Dearle

al@cs.adelaide.edu.au

Department of Computer Science
G.P.O. Box 498, Adelaide
South Australia 5001
Australia

Abstract

This paper describes the use of a persistent store to support incremental system construction. A single example is elaborated throughout the paper to introduce elements of the incremental construction mechanism. The essence of the technique is to permit assignment of executable program fragments to typed locations in order to change the behaviour of a program under construction.

The technique described in this paper relies upon the existence of three features: a persistent store, updatable locations and first class executable values. The examples in this paper use the persistent programming language Napier88, but any language with these features could be used to construct such a system.

1. Introduction

In many programming environments, the compilation units are syntactic entities which typically contain some data and a set of procedures (or functions) which provide access to that data. This approach is exemplified by packages in Ada [1] and Object Oriented Design Methodology [6]. This paper describes a more flexible scheme which permits incremental system construction and a high degree of component reuse. In the technique described in this paper, modules are more akin to operating system directories than the syntactic units to which most of us are accustomed.

In order to support a technique such as the one described in this paper, an object store is required as a repository for modules. This requirement has been recognised by other researchers in the field, for example, Boehm [5] cites the provision of a project master data base or persistent object base as one of the important features distinguishing an Integrated Project Support Environment (IPSE) from a collection of ad hoc tools. The need for a uniform, homogeneous object storage facility is also identified by the Portable Common Tools Environment (PCTE) [16] and the Ada Apse designers [14].

Programming languages normally have little support for the maintenance of long term data. The only concession usually made is the provision of a file data type. Therefore, the programmer is faced with the task of mapping data onto long term storage which is usually provided by the file system or DBMS. The mapping of data between long and short term

storage is expensive, both in terms of programmer design time and program run time. In 1978, Atkinson [2] recognised this problem and isolated a property of data known as *persistence*. Persistent programming is a relatively new paradigm that makes data intensive application programming significantly easier. The idea behind persistence is a simple one: data in a system should be able to persist (survive) for as long as that data is required. Orthogonal persistence means that all data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. In other words, the right for data to survive for a long (or short) time is independent of the type of data. Programs manipulating data do so in a uniform manner, whether the data is short or long lived.

Software development environments require the storage of large amounts of fine grain program components and the relationships between them. The uniform, homogeneous object storage facility provided by a persistent store makes an ideal vehicle for the construction of IPSE's [12].

Parnas [15] states that any part of a program which could conceivably be reused should form a module. The advantages of making the unit of change (a module) small may be observed in systems such as Smalltalk-80 [9] in which large amounts of fine grain code exists which may be (and is) easily reused. In the examples in this paper, procedures† that are first class data objects [4] are used as the units of compilation and reuse. It should be noted that although procedures are used in this paper as the unit of reuse, any executable value which may be assigned to a location may be used in their place.

Making procedures first class data objects permits them to be treated like any other data type; in particular, it allows procedural data to be stored in the persistent object store. Procedures also have the advantage that they are relatively small and self contained. First class procedures may also be used to provide encapsulation and protection of data. The power of traditional library facilities, modules and generics are subsumed when first class procedures are combined with persistence and a powerful type system that includes parametric polymorphism. In this paper, the programming language Napier88 [13] is used as the vehicle to describe the incremental system construction methodology. It has first class procedures, assignment, orthogonal persistence and a polymorphic type system.

This paper describes an incremental method of system construction. A simple example is introduced and used throughout the paper to illustrate the techniques involved. Although the example is extremely simple, the method of system construction scales without loss of generality. The paper first describes how the behaviour of programs may be changed using assignment, L-value binding and first class procedures. The next section demonstrates the use of a persistent store as a repository for components under construction. The construction of parameterised components is the subject of the next section. Finally, issues of type evolution and sealing are discussed.

---

† In this paper the word procedure will be used synonomously with the word function.

1.1 The language Napier88

Before presenting an example, it is expedient to make a few comments about the language Napier88 which is used in the examples throughout this paper. In line with the principle of data type completeness [10], all Napier88 data types share the same civil rights. These are: the right to be declared, the right to be passed to and be the result of a procedure, and the right to the full spectrum of persistence. Since much of this paper is concerned with binding mechanisms, it is prudent to spend some time describing the binding mechanisms provided by Napier88. These mechanisms are described in more detail in [11].

In Napier88, a binding may be instantiated anywhere in a block (as opposed to at the beginning of a block in languages like Pascal). Bindings are always initialised in Napier88, it is impossible to create a binding without specifying an associated value. The language allows two different kinds of bindings to be created: constant bindings and variable bindings, both of which may be instantiated with a value which is unknown at compile time. Conceptually, all Napier88 bindings are L-value (location) bindings. That is, when a name is used, the appropriate value is fetched from the location denoted by that name. In Napier88, bindings are introduced using the keyword "let". For example,

let a = 7

creates a constant binding from the name *a* to the value 7. The declaration enriches the scope in which it is made. After the declaration the name *a* may be used. Similarly, a constant binding may be created to a statically unknown value as follows:

let b = factorial(42)

In this case it is impossible to know the value to which b is bound until the procedure *factorial* has returned a value (at run time). Variable bindings are syntactically distinguished from constant bindings by the use of ":=" rather than "=" during their creation. Thus,

let a := 7

denotes a binding between the name *a* and a location which initially contains the value 7. Variable locations may be updated by assignments such as:

a := 3

Similarly, the programmer may create constant or variable bindings to procedure values. For example, the following creates a variable binding to the *add* procedure:

let add := **proc**(a, b : int → int) ; a + b

2. An example

An example will now be introduced and subsequently elaborated. The example is used to illustrate two different points: the power of the language Napier88 and the incremental

system construction architecture. In the example, four procedures are defined and used: a procedure called *inc* to increment a number, one to decrement a number called *dec*, another to add two positive numbers called *add*, and finally a procedure called *double* which doubles a number; these procedures are shown in Figure 1.

```
let inc := proc( i : int → int ) ; i + 1
let dec := proc( i : int → int ) ; i – 1

let add := proc( a, b : int → int )
begin
        while b > 0 do
        begin
                a := inc( a )
                b := dec( b )
        end
        a       ! return a as the result.
end

let double := proc( j : int → int ) ; add( j, j )
```

Figure 1: A small Napier88 program

In Figure 1, the name *double* is bound to a location containing a procedure closure. This closure (the value of a procedure) is bound to a location containing the procedure closure for *add*. The procedure *add* is itself bound to two other locations, namely the locations containing the procedure closures for *inc* and *dec*. These bindings may be diagrammatically represented as shown in Figure 2:

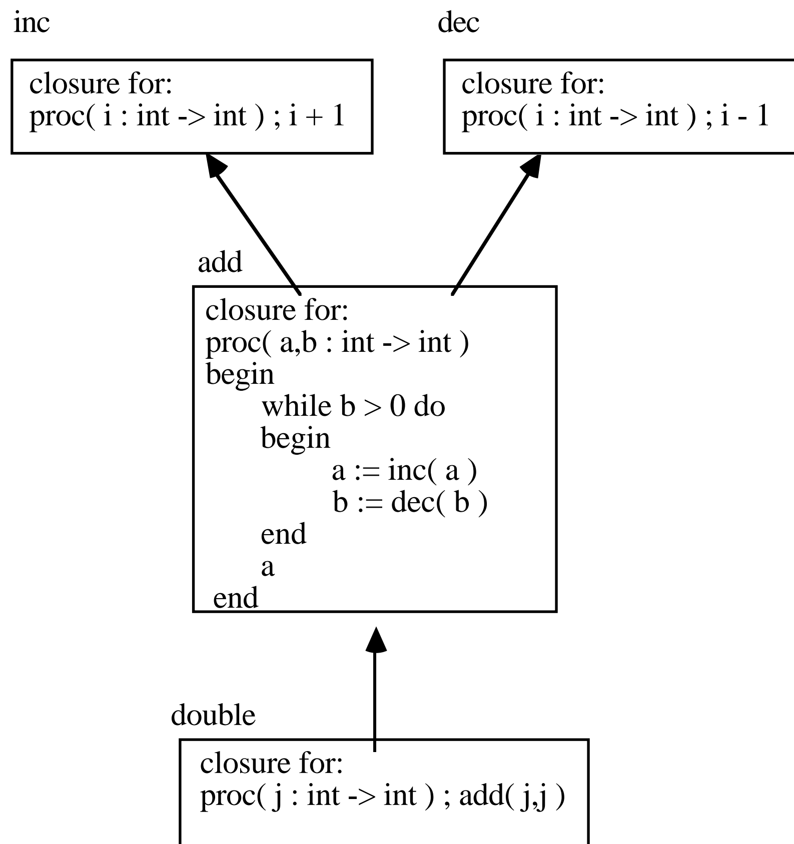

Figure 2: The binding graph for the program shown in Figure 1.

3. Changing program behaviour using assignment

In the example shown in Figure 2, the components are bound using L-value bindings. That is, the value of the procedure stored at the location called *double* is bound to the location containing the procedure *add* and not the value of the *add* procedure. Similarly, the *add* procedure is bound to the locations containing the *inc* and *dec* procedures. This binding mechanism is extremely important since if the values of procedures had been bound using direct value bindings no evolution could take place in the system.

As the system stands, evolution may take place. For example, the procedure value contained in the location denoted by *add* may be replaced with a more efficient one which does not use *inc* or *dec*. Since a location exists containing *add*, this may be achieved using a simple assignment† as shown in Figure 3.

---

† Provided that the procedure being assigned and the location are of the same type. This issue will be addressed later in the paper.

```
let inc := proc( i : int → int ) ; i + 1
let dec := proc( i : int → int ) ; i – 1

let add := proc( a, b : int → int )
begin
        while b > 0 do
        begin
                a := inc( a )
                b := dec( b )
        end
        a       ! return a as the result.
end

let double := proc( j : int → int ) ; add( j, j )

add := proc( a, b → int ) ; a + b
```

Figure 3: Assignment to a location containing a procedure value.

Since the procedure denoted by *double* is bound to the location containing *add* and not the value of the procedure *add*, the procedure denoted by *double* will reflect the changes made to *add*. The value that was bound to the location *add* is still in existence. Any programs bound to that value rather than the location will continue to use the old code. The binding graph for the system after the assignment is shown in Figure 4.

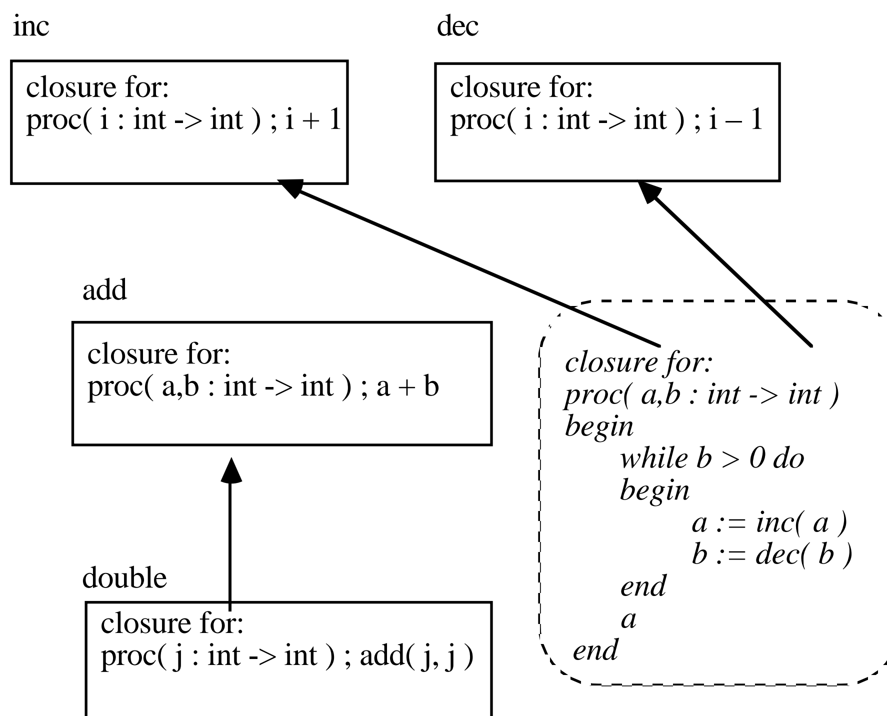

Figure 4: The binding graph for the program shown in Figure 1 after an assignment.

Thus a system may be evolved using first class procedures and bindings to locations. Traditionally, the program fragments shown above would be encapsulated within some syntactic mechanism such as a file or a module. In order to allow evolution to occur, some general mechanism is required to allow the locations containing the program fragments to be updated. The incremental construction technique described in this paper proposes the persistent store as a repository for these bindings.

4. Using the store as a module repository

The persistent store serves to permit locations containing program fragments to be accessed during the program construction process. In Napier88 this mechanism is provided by a data type called *environment* [8]. Environments are collections of bindings which may be extended or contracted under user control. All environments belong to the same infinite union type, denoted by the type name **env**. For each binding contained in an environment, the Napier88 system maintains an identifier, a value, a type and a constancy indicator. By manipulating the bindings in environments, the user can control the name space. To create an environment, the standard procedure *environment* is used, it has the following type:

proc(→ env)

Calling this procedure creates an environment containing no bindings. Adding a binding to an environment is the dynamic analogy of adding a declaration (binding) to a scope level. Bindings are therefore added to an environment by making a declaration. For example,

let firstEnv = environment( )
in firstEnv **let** inc := **proc**(i : int → int) ; i + 1

creates an environment, denoted by *firstEnv*, and then places the binding associated with the name *inc* in it. In this example, the environment records the identifier *inc*, the value "**proc**(i : int → int) ; i + 1", the type "proc(int → int)" and the fact that the binding is updatable. Unlike normal declarations, environment declarations do not enrich the local scope; that is the name *inc* is not in scope following the environment declaration. Instead the declaration is added to the environment denoted by *firstEnv*. To use the bindings contained in an environment, a projection statement, known as a **use** clause, is invoked which projects a binding from an environment into a local scope. For example,

use firstEnv **with** inc : **proc**(int → int) **in** .... inc(7) ....

introduces the value bound to *inc* in the environment *firstEnv* into the local scope. The binding used is an L-value binding. This means that, if an assignment is made to *inc* in the body of code following the **in**, the value that is bound to the environment will be changed. Such a projection invokes a dynamic check to ensure that a binding with name *inc* exists in the environment and that the projected value matches the expected type. This is one of two places in Napier88 where dynamic checking is performed – in all other cases checking is performed statically. The environment **use** clause need only specify the particular bindings that are to be used from the environment.

4.1. Environments and the persistent store

Environments are used by convention to impose structure on the persistent store. In accordance with the concept of orthogonal persistence [3], all data objects in Napier88 may persist. For each incarnation of the Napier88 object store, there is a root of persistence which may be obtained by calling the (only) predefined procedure called *PS* which has the type:

```
proc( → env )
```

Thus, the distinguished root of the persistent store graph is of type **env**. Every value stored in the persistent store is reachable from this environment. A set of standard bindings is provided with every Napier88 persistent store. One of these bindings contains the procedure used earlier to create environments. It is bound to the identifier *environment* in the root environment. The following program illustrates the use of this procedure:

```
let ps = PS()
use ps with environment : proc ( → env ) in
begin
        let newEnv = environment()
        in newEnv let inc := proc( i : int → int ) ; i + 1
end
```

Figure 5: Using the Napier88 environment procedure.

The program shown in Figure 5 binds the root of persistence to the local identifier *ps*. The **use** clause binds the local identifier *environment* to the environment creation procedure in the root environment. Inside the body of the **use** clause, this procedure is called to create a new (empty) environment. Finally, a binding denoted by *inc*, is made in the newly created environment. The reader should note that the environment bound to *newEnv* is not yet persistent. Objects that persist beyond the activation of the program unit that created them are those which the user has arranged to be reachable from the root of persistence. To determine this, the system computes the transitive closure of all objects starting with the root. Thus, in order to place objects in the persistent store, the user has to alter or add bindings that can be reached from the distinguished root. In order to make the environment *newEnv* persist, the user may write:

```
let ps = PS()
use ps with environment : proc(→ env ) in
        in ps let newEnv = environment()
```

Figure 6: Adding bindings to the root environment.

which adds the binding denoted by *newEnv* to the root environment. Once the environment *newEnv* is made persistent, the procedures *inc* and *dec* may be placed into it as shown in Figure 7 below:

```
use PS() with newEnv : env in
begin
    in newEnv let inc := proc( i : int → int ) ; i + 1
    in newEnv let dec := proc( i : int → int ) ; i - 1
end
```

Figure 7: Adding procedure bindings to *newEnv*.

This program binds the procedures *inc* and *dec* to the environment *newEnv* which is located in the root environment. Since the environment *newEnv* is persistent and the procedures denoted by *inc* and *dec* are reachable from *newEnv*, they too are persistent. The procedure *add* may now be written to use these procedures as shown in Figure 8 below:

```
use PS() with newEnv in
use newEnv with inc, dec : proc( int → int ) in
    in newEnv let add :=  proc( a, b : int → int )
                          begin
                              while b > 0 do
                              begin
                                  a := inc( a )
                                  b := dec( b )
                              end
                              a       ! return a as the result.
                          end
```

Figure 8: Adding *add* to *newEnv*.

In Figure 8, the procedure denoted by *add* is bound to the locations containing the procedures *inc* and *dec*. Any consequent changes to the locations bound to these names will be reflected in the semantics of the procedure denoted by *add*. The binding graph for the code in Figure 8 is shown in Figure 9 below.

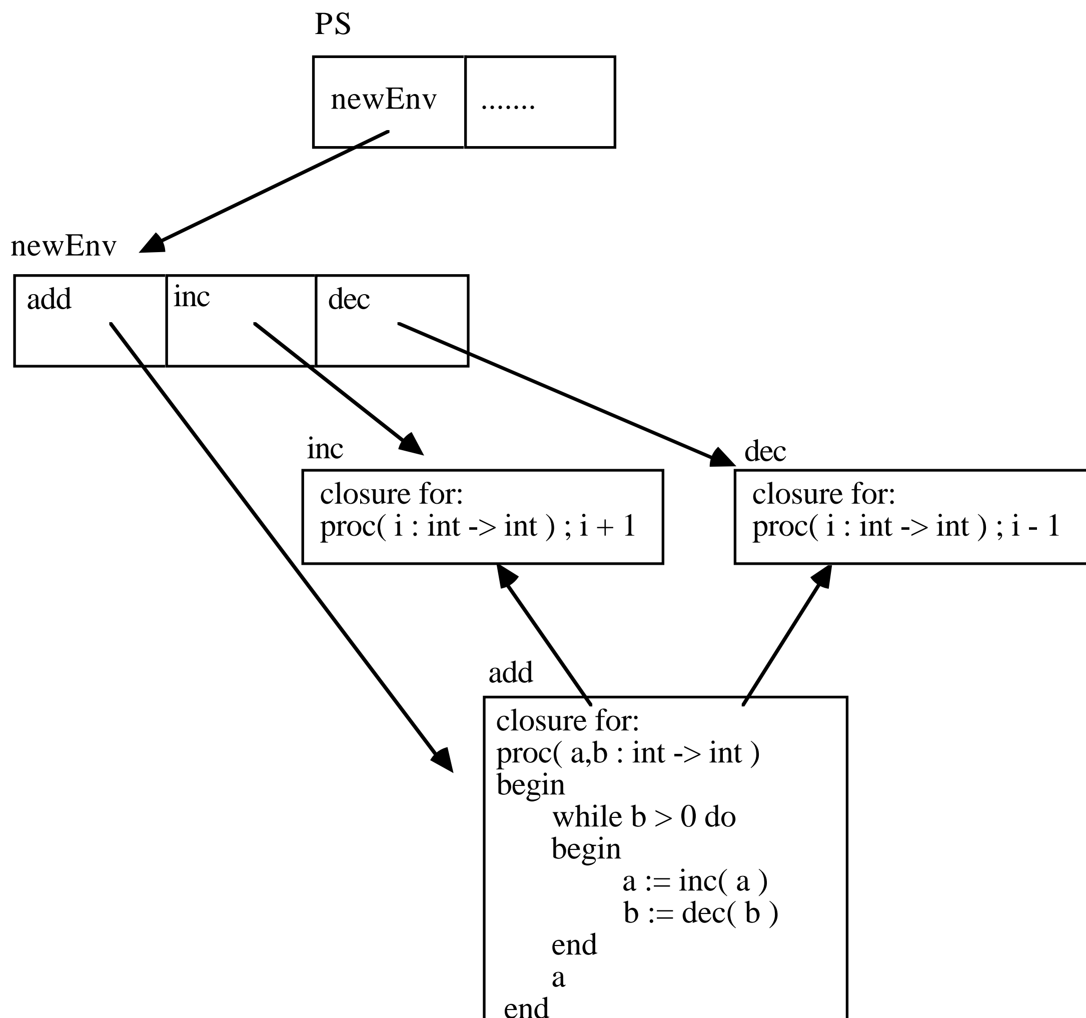

Figure 9: The binding graph corresponding to Figure 8.

Notice that it is essentially the same graph as that shown in Figure 2. This shows that using the binding mechanisms found in environments, the block structure of Napier88 may be dynamically constructed. The difference is that the individual program elements are no longer trapped inside syntactic enclosures such as files or modules. Instead they are contained in a type secure object store where they may be reused. Since the individual procedures may be separately compiled and bound, a framework to support evolution and reuse is beginning to emerge.

5. Constructing parameterised components

It has been shown that procedures may be placed into the persistent store and their behaviour modified by altering the contents of the locations to which they are bound. However, sometimes a developer wishes to reuse procedure code but does not wish to reuse the statically bound environment of that procedure. For example, consider the procedure *double* shown in Figure 1. It uses a procedure called *add* to add two integers together. The code of *double* could be reused independently of the actual instance of the procedure *add* bound to it. It is easy to imagine two instances of *double*, one bound to the procedure *add* shown in Figure 8 and another bound to the following version of *add*:

```
let add = proc( a, b : int → int ) ; a + b
```

The ability to selectively reuse code without being forced to share the same static referencing environment may be achieved by parameterising the system building components. In Napier88 this requirement is satisfied by the use of generators.

Generators are procedures which produce other procedures as a result of their application. Generators therefore utilise the power of first class procedures and will be introduced using a new example. Consider the piece of code shown in Figure 10 below; the procedure denoted by *putGet* returns the value of the location denoted by *register*. The integer passed as a parameter to *putGet* is saved in this location and is returned on the next call. Note that *register* is a free variable with respect to the procedure *putGet*. This means that the variable *register* will be shared by all instantiations of the procedure.

```
let register := 0

let putGet = proc( param : int → int )
begin
	let temp = register
	register := param
	temp
end
```

Figure 10: The *putGet* procedure.

Since the procedure uses a single shared store location, it cannot reliably be used to save values by a number of different procedures. This may be overcome by wrapping the procedure in a generator as follows:

```
let putGetGen = proc( initial : int → proc( int → int ) )
begin
    let register := initial              ! this is encapsulated

    proc( param : int → int )            ! this is the procedure returned
    begin
        let temp = register
        register := param
        temp
    end
end
```

Figure 11: A *putGet* generator procedure.

Every program wishing to use *putGet* may now do so safely by calling *putGetGen* in order to obtain a *putGet* procedure with its own store. The location denoted by *register* is a free variable with respect to the (anonymous) *putGet* procedure but bound with respect to the generator. As shown in Figure 12 below, every application of the generator yields a new *putGet* procedure with its own encapsulated store in the form of the variable *register*. Although a new *putGet* value is produced on every application of *putGetGen*, all the instances of *putGet* share the same code; in this way procedures may share code without having to share state.

```
let putGet1 = putGetGen( 10 )
let putGet2 = putGetGen( 3 )

writeInt( putGet1( 13 ) )        ! writes out 10
writeInt( putGet2( 5 ) )         ! writes out 3
writeInt( putGet1( 6 ) )         ! writes out 13
```

Figure 12: Two uses of the *putGet* generator.

5.1. Generators and code reuse

First class procedures may be used to provide a mechanism which allows the sharing of code without the sharing of state. The notion of state may be extended to include all the values bound to a particular procedure closure, including bound procedural values. When first class procedures are considered, these include bound procedural values. Applying this technique to the procedure *add* shown in Figure 1 yields the following:

```
let addGen = proc( inc, dec : proc( int → int ) → proc( int, int → int ) )
begin
	proc( a, b : int → int )	! this procedure is the result of addGen
	begin
		while b > 0 do
		begin
			a := inc( a )
			b := dec( b )
		end
		a	! return a as the result.
	end
end
```

Figure 13: A generator for *add*.

The code of procedure *add* may now be bound to any two arbitrary procedures of type *proc(int → int)*. The *add* procedure may still be type checked statically by the compiler, although the binding of the actual procedure values, *inc* and *dec*, has been delayed until run-time. Since Napier88 has call by value semantics, the procedure instances generated by *addGen* have direct value bindings to the actual procedures *inc* and *dec*. Thus one of the advantages discussed above is lost, namely the instances of *inc* and *dec* bound to an instance of *add* may not be updated by simple assignment. Some modification of the technique is necessary if this desirable property is to be retained.

5.2. Generators and environments

Combining the techniques shown in Figures 8 and 13 produces a powerful and flexible architecture. If environments, rather than procedures, are passed as parameters to generator procedures, the generated code may make L-value bindings to the procedure values stored in those environments. This is best illustrated by the example shown in Figure 14 below:

```
let addGen = proc( incEnv, decEnv : env → proc( int, int → int ) )
begin
        use incEnv with inc : proc( int → int ) in
        use decEnv with dec : proc( int → int ) in
        begin
                proc( a, b : int → int )        ! this is the result of addGen
                begin
                        while b > 0 do
                        begin
                                a := inc( a )
                                b := dec( b )
                        end
                        a                       ! return a as the result.
                end
        end
end
```

Figure 14: A generator for *add* which uses environments

In the example shown in Figure 14, the generator procedure is passed two environments as parameters and returns an *add* procedure. The *add* procedure contains bindings to the procedure locations named *inc* and *dec* contained within the two environments. Thus any assignments to the locations containing *inc* and *dec* will be reflected in any procedure generated by *addGen* and bound to those locations.

If an environment called *newEnv* has been initialised as shown in Figure 7, the generator given above may be called as in Figure 15.

```
use PS() with newEnv : env in
begin
        in newEnv let add = addGen( newEnv, newEnv )
end
```

Figure 15: Using *addGen* to generate an instance of *add*.

This call will result in an instance of the *add* procedure being placed in the environment denoted by *newEnv* and bound to the locations containing *inc* and *dec* in the same environment as illustrated in Figure 16.

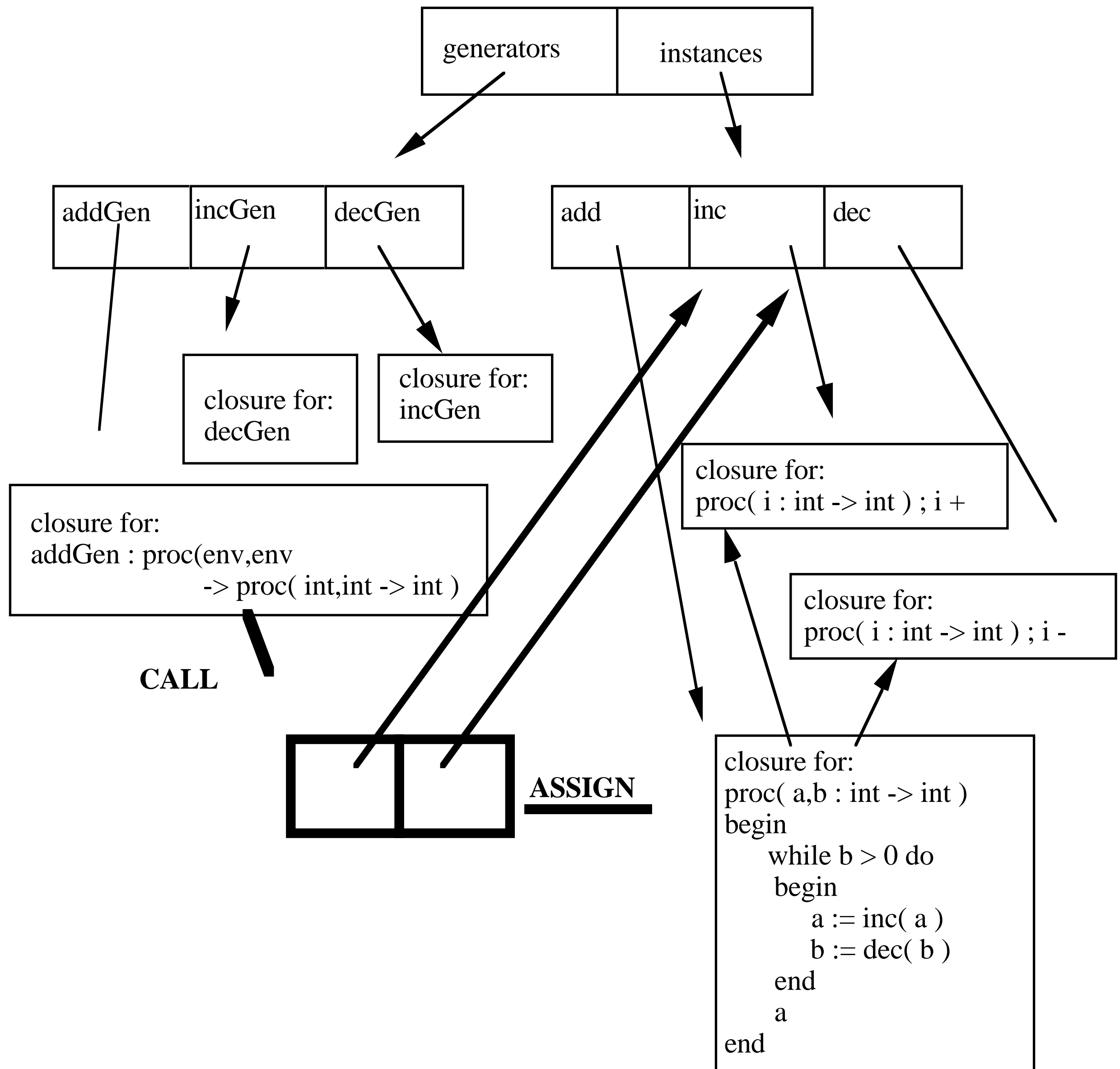

Figure 16: Generating the *add* procedure

Note that the application binding graph is identical to that shown in Figure 9. However, using this technique, many different versions of *add* may be generated which all share the same code but have different referencing environments and hence different behaviour. In general, the construction of an instance of a procedure such as *add* involves four steps:

1. Extract the generator procedure from the store.

2. Extract the environments containing the desired parameters from the store.

3. Apply the generator procedure with the appropriate environments as parameters.

4. Assign the resulting procedure to the appropriate ‘slot’ in the instance graph.

5.3. Generator, instance and application graphs.

In Figure 14, two different levels of procedures are being manipulated: *generators* like *addGen* and *instances* like *add*, *inc* and *dec*, which may or may not have been created by generators. In general, an application is composed of procedure instances. Some infrastructure is required to manage the procedures in the store. The strategy currently adopted is to maintain three graphs. The first two are isomorphic, one containing the generators, the other the instances. These are explicitly maintained using environments as a structuring mechanism. The third graph is held implicitly within the system and is the binding graph of the application under construction. It is formed from procedure values bound to the typed locations containing other procedure values. The three graphs will be referred to as the generator , instance and application graphs. They all reside in the persistent store and are constructed using the mechanisms described earlier. A good analogy is that the generator and instance graphs represent the scaffolding supporting the construction of the application, whereas the application graph contains the bindings in the actual application under construction. The graphs are illustrated in Figure 17 below.

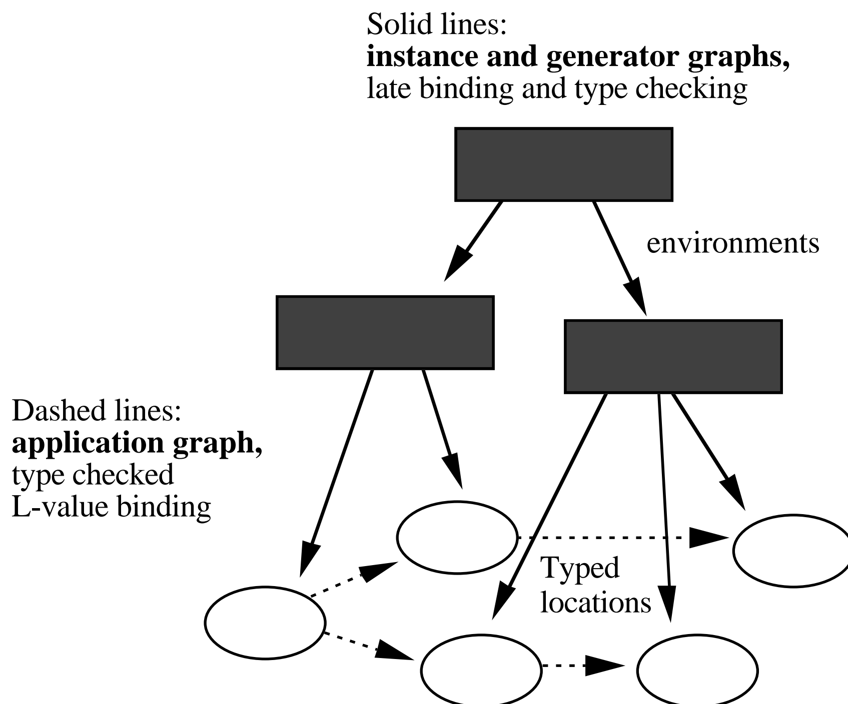

Figure 17: Generator, instance and application graphs.

The three graphs use different binding and type checking regimes. The generator and instance graphs use late binding and type checking to provide a high degree of flexibility. The store locations containing the procedure values are highly accessible and their contents easily modified using assignment. This is made possible by the provision of an access path to those locations through the instance graph. The application graph is bound and type checked when the procedure values are placed in the instance graph. Usually this will be at the time that generators are executed. This means that no type checking or binding is necessary at application execution time.

6. Type evolution and sealing.

In the discussion that has taken place, it has been assumed that procedures are only replaced by procedures of the same type. This is an unrealistic requirement, often during the development of a system the signatures of modules change as new requirements are made of them. Therefore, for the technique described in this paper to be of any practical use, it must deal with changes in the types of bound procedures without requiring a total system rebuild.

Fortunately the problem may be solved using a modification of the method described above. Consider the binding graph shown in Figure 18 below in which the ovals represent L-values which contain procedure values. All the procedural values in the system bind directly or indirectly to the dark location in the centre of the diagram. Should the type of this location change, the values contained in those ovals shaded in a lighter colour must be recompiled. This is necessary because any change to the type of the dark location will render their specification incorrect. Even though these procedures must be changed, they will not change type; consequently, the required changes may be implemented using the mechanisms described earlier. The values stored in the white locations at the outside of the binding matrix bind to the lightly shaded locations rather than values. They will be unaffected by any changes to the centre location.

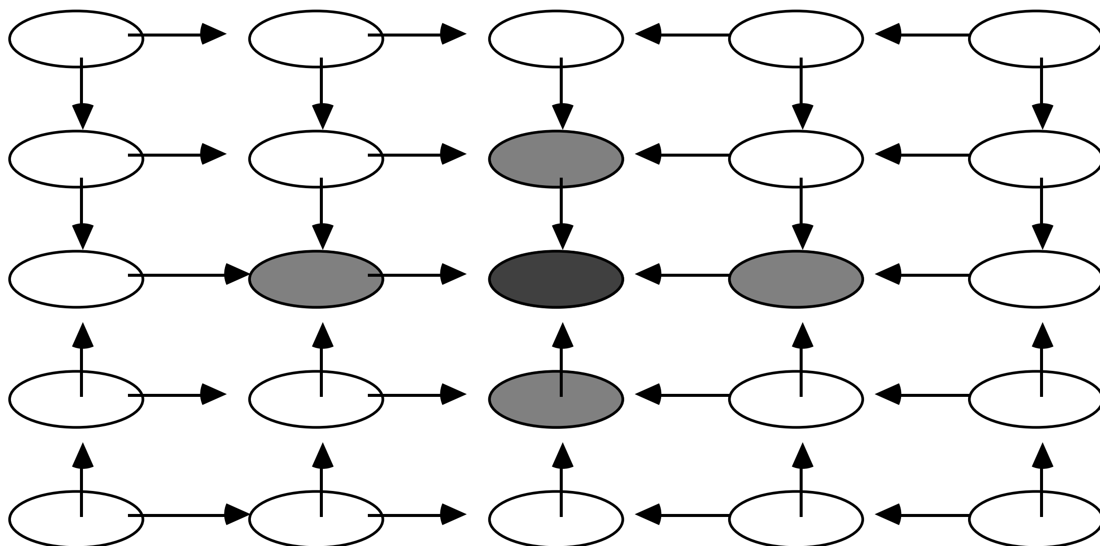

Figure 18: A binding matrix.

The mechanism to change the type of any procedure is therefore simple:

1. The location containing the old value is removed.

2. A new location is created of the appropriate new type.

3. The procedure source must be edited and the new compiled procedural value assigned to the new location (in practice, steps 2 and 3 may be merged).

4. The procedures which make use of the procedure whose type has changed must be altered, compiled and assigned to their original locations in the binding matrix.

This amounts to making a hole in the application binding matrix and then darning in a new piece of code. In most cases, the majority of the application will be unaffected and therefore will not require any recompilation or rebinding.

6.1. Sealing

When a designer is satisfied with a system under development, it must be capable of being protected from accidental or malicious change; this task is sometimes known as *sealing*. [7] Using the techniques described in this paper, sealing is a relatively trivial matter. All that is required is the removal of the access path to the locations containing the application whilst retaining at least one entry point to the program. Since the Napier88 system defines persistence by reachability, the locations containing the components of the system are retained provided that they are reachable from the persistent root. Since the locations are reachable from the program entry point(s) they will persist, even though the programmer can no longer name them.

This technique will be illustrated by example. Suppose that the system shown in Figure 9 exists and the programmer wishes to seal it, preserving only the entry point called *add*. This may be achieved as shown in Figure 19 below.

```
let ps = PS()
use PS() with newEnv : env in
        use newEnv with add : proc( int, int → int ) in
                in ps let newAdd = add

drop newEnv from ps
```

Figure 19: Sealing a system.

In this program, the first line obtains the root of persistence. The next three lines make a binding from the root environment to the location denoted by *add* in the environment *newEnv*. Lastly, the environment denoted by *newEnv* is removed from the environment *ps*. Once this program is executed, access to the environments containing the constituent parts of the application is removed and so too is the capability to change the application. All that remains is the single entry point called *newAdd*. The application graph still exists, all that has changed is that the scaffolding and the locations containing the procedures are no longer accessible by the programmer. As stated earlier, these locations persist since they are reachable from the value stored in the location denoted by *newAdd*. The situation after the code in Figure 19 has been executed is illustrated in Figure 20 below. Note that this system is essentially the same as that shown in Figure 2.

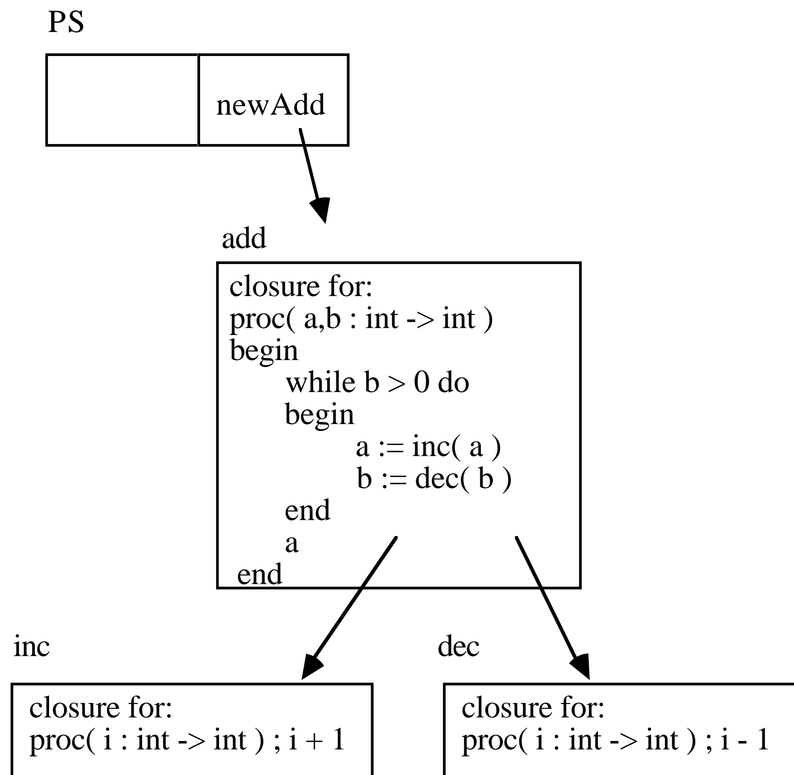

Figure 20: A sealed system.

7. Conclusions

The incremental system building technique described in this paper utilises three features of the Napier88 programming language: a persistent store, updatable locations and first class procedures. Although the techniques have been described in terms of Napier88, any language with these features could be used to construct such a system. Indeed, although the evolution mechanism has been described using first class procedures, any value may be used as the unit of system construction. The technique can be used with locations of many different types within a single application.

The essence of the technique is to construct programs 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. These operations may be performed at the time that the generators are executed which is analogous to link time in traditional programming systems. This is facilitated through two types of binding graph – a strongly typed L-value binding mechanism internal to the application under construction and a dynamically typed access mechanism which provides the programmer with access to the locations containing program fragments.

The use of generators to provide a parameterisation mechanism is also of great utility. Generators permit executable code to be shared without the need to share a referencing

environment. This technique aids experimentation and evolution without any loss of run time performance.

Using this technique, all code is potentially reusable with essentially no associated run time overhead. Furthermore there is a choice of levels at which code may be reused. In addition to the reuse of source code, as commonly practiced in traditional programming environments, bound executable instance code may be reused or, if a different configuration is required, the generators may be used to produce new instances.

The technique is designed to facilitate the construction of systems using an evolutionary approach. The user may add new modules to an existing program at any time. These may be implemented independently of the 'main' application and bound later – perhaps after unit testing. Existing programs may be updated with the same ease. It has been shown that it is possible to seal systems which are constructed using this technique, at low cost and with little disruption to the application that has been constructed.

8. Current status and future directions

The technique has been to construct a compiler for Napier88 written in the language. This is an application consisting of more than 10,000 lines of code split between about 120 separate components. It has also been used to construct window management, editing and browsing tools for the language. Current work is attempting to unite all these components into an integrated programming environment written entirely in Napier88 which will support the construction of programs using the architecture.

Acknowledgments

This work is supported by grants from The Defence Science Technology Organisation of Australia (DSTO), The Australian Research Council and by The University of Adelaide. This paper benefits from many useful discussions with colleagues at Adelaide University, The University of St Andrews, Sydney University and Flinders University of South Australia.

References

1. "The Programming Language Ada Reference Manual. ANSI/MIL-std-1815a-1983.", Springer Verlag, 1983.

2. Atkinson, M. P. "Programming Languages and Databases", *Fourth IEEE International Conference on Very Large Databases*, IEEE, pp. 408 - 419, 1978.

3. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, vol 26, 4, pp. 360 - 365, 1983.

4. Atkinson, M. P. and Morrison, R. "Procedures as Persistent Data Objects", *Transactions on Programming Languages and Systems*, vol 7, 4, ACM, TOPLAS, pp. 539-559, 1985.

5. Boehm, B. W. "Understanding and Controlling Software Costs", *Information Processing 86*, pp. 703, 1986.

6. Booch, G. "Object Oriented Design", Benjamin-Cummings, 0-0853-0091-0, 1991.

7. Cardelli, L. "Typeful Programming", DEC, 45, 1989.

8. Dearle, A. "Environments: A Flexible Binding Mechanism to Support System Evolution", *Proc. 22nd Hawaii International Conference on System Sciences*, vol II, Hawaii, pp. 46-55, 1989.

9. Goldberg, A. and Robson, D. "Smalltalk-80: The language and its Implementation", Addison Wesley, 1983.

10. Morris, J. H. "Protection in programming languages", *CACM*, vol 16, 1, pp. 15-21, 1973.

11. 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.

12. Morrison, R., Bailey, P. J., Brown, A. L., Dearle, A. and Atkinson, M. P. "The Persistent Store as an enabling technology for Integrated Support Environments", *8th International Conference on Software Engineering*, pp. 166-172, 1985.

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

14. Office, K. C. W. G. f. t. A. J. P. "Common APSE Interface Set. Version 1.1", Ada Joint Program Office, 1983.

15. Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol 15, 12, pp. 1053-1058, 1972.

16. Thomas, I. "PCTE Interfaces: Supporting Tools in Software-Engineering Environments", *IEEE Software*, pp. 15-23, 1989.