Persistent Program Construction through Browsing and User Gesture with some Typing

Alex Farkas, Alan Dearle

Department of Computer Science, University of Adelaide
Adelaide, Australia

Graham Kirby, Quintin Cutts, Ron Morrison, Richard Connor

Department of Computational Science, University of St. Andrews
St. Andrews, Scotland

Abstract

One method of evaluating programs is for them to be prepared as self contained pieces of source, then compiled, linked and executed. The last phase may involve binding to and manipulating persistent values. When the persistent store is supported by a user interface, the program construction can be augmented by the use of tokens as denotations for persistent values. That is, the manipulation of the persistent store by gesture, for example by an iconic interface linked to a mouse, can be used to provide tokens for persistent values. These tokens can be resolved to provide bindings at run-time, compile-time, program construction time or any mixture of these.

In this paper the main styles of token resolution are described in terms of their influence on the persistent program evaluation. This is done in tandem with a description of an example user interface required to support these new styles of persistent programming. We note that other modern user interfaces, such as OpenLook and the Macintosh Programming Environment also allow the manipulation of files by user programs and by gesture. The difference here is that the technique is uniform and that the persistent store is strongly typed with a greater variety of types.

Two prototype versions of these facilities have been implemented for the language Napier88.

Keywords: browser, binding, object oriented, persistence, programming environment.

1 Introduction

Most modern applications systems make use of an iconic interface linked to a mouse controller. This allows many operations on data to be described by user gesture rather than by the typing of a textual command. The advantages of this style of interface are well understood and documented.

One area this style of interaction has not generally pervaded is the activity of software construction. Some limited user gesture interaction may be possible, but in general programs consist of a flat textual representation of code. Mostly this code is typed by a programmer, although some systems provide support for a limited amount of automatic code production according to a programmer's description at a higher level of abstraction.

The advantages of user gesture can, however, be incorporated into the construction process for programs which make use of persistent data. Such programs normally contain textual code which describes an access path to data within the persistent store. As an alternative it may be envisaged that, rather than writing the textual form of this code, a programmer may be provided with iconic tools which allow the browsing of the persistent store, and then a particular value within the store may be indicated by some mouse gesture. This method of interaction is the topic of this paper.

The essence of the method is to include a persistent store browser, along with the notion of a token, as a part of the program construction environment. A value of interest encountered during browsing may be denoted or "tagged" by the use of a token. This token may then be used to construct a reference to the value it denotes.

The methodology of programming with tagged values is described in a practical manner, by the description of a program construction environment. The purpose of this is to make clear the potential use of such a mechanism in a practical context. Two prototype versions of such a program construction environment [1,4] have been constructed for the language Napier88 [6].

Three different schemes are described in this paper, the difference between them being the time at which persistent data is bound into the program being constructed. None of these schemes is intended to be used in isolation; indeed it is envisaged that a judicious mixture of the three different binding times will be of use in a complex application.

In the first scheme, programs contain code to locate data and the data is dynamically bound during the execution of the program in the normal fashion. In this scheme, the browser provides an interface to the programmer through which the data to be bound may be located. This in turn enables the programmer to construct an appropriate computation to locate the data.

The second scheme allows values from the persistent store to be bound during compilation, rather than execution. To achieve this the compiler must also be included in the execution environment. The meaning of a token within a program is not resolved by expansion into high-level code; instead when the program is compiled the value denoted by the token is resolved by the compiler and bound directly into the executable code. The meaning of a program in this system is dependent upon the environment in which it is compiled, and is no longer self-contained.

The last scheme is known as hyper-programming. Hyper-programs are constructed in a similar way to the other schemes described. Rather than the tokens being resolved by the compiler, however, the user may indicate that the tagged value itself should be included directly within the high-level program code. This requires a relatively sophisticated program editing tool, as programs may no longer be represented as flat textual structures. It may thus be seen that hyper-programs bear a similar relation to normal programs as hyper-text does to normal text.

Section 2 introduces an example Napier88 program which will be used throughout the rest of the description. Sections 3,4 and 5 describe the use of the three different binding styles to construct the example, and Sections 6 and 7 conclude with the possibilities of future research in this area.

2 An example

2.1 An example persistent store

A Napier88 persistent store consists of a graph of values connected by pointers and may be accessed from a single point known as the *persistent root.* The root of a persistent store may be accessed by executing the predefined Napier88 function *PS*, which returns a dynamically extensible collection of bindings known as an *environment* [2]. An important property of environments is that they enable the programmer to make bindings to typed locations as well as values.

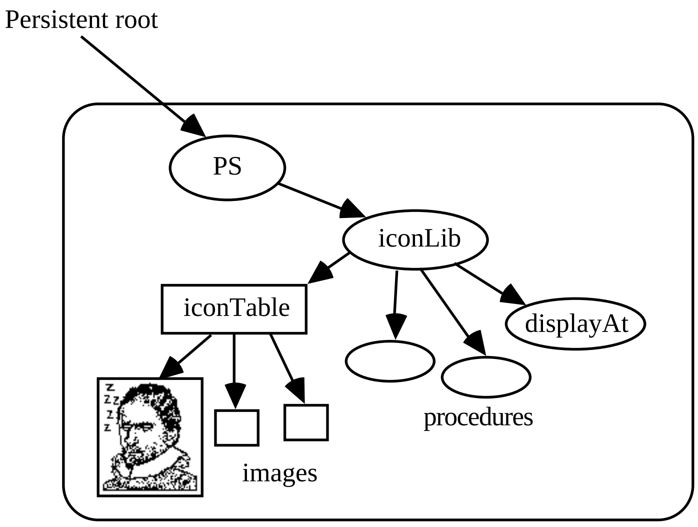


Figure 1: Conceptual view of a simple icon management system.

To illustrate the use of a persistent store, consider as an example a simple icon management facility structured as follows. In the root environment of the persistent store another environment called *iconLib* containing a table of icons, *iconTable*, and a number of procedures has been constructed. These procedures operate on icons which are of type **image**, a Napier88 data type which consists of a rectangular array of pixels. The *iconTable* consists of a table of icons indexed by a unique name in the form of a string. The structure of this store is shown in Figure 1.

In order to manipulate values in a persistent store, a Napier88 program must first bind to those values. In order for binding to occur, the value or values being bound must be located and type checking must take place to ensure that they have the type expected by the program. In the programming environment described in this paper, binding may take place at three different times during a program's life cycle:

- during program execution: the program binds to values at run-time,

- during compilation: binding takes place as the source code for a program is compiled, and
- during program construction: binding occurs as the source code for the program is constructed.

The sections which follow describe in more detail the manner in which each style of binding may be achieved in a persistent programming environment. First we describe an application that will be used as an example throughout the paper.

2.2 An example application

To demonstrate the different programming paradigms, the following application will be used as an example. A procedure called *wallPaper* shall be constructed which behaves as follows: when invoked, the procedure displays the icon associated with the name "John Napier" in the *iconTable*. The procedure displays the icon a number of times so that it "wallpapers", i.e. completely covers, the screen. In order to achieve this, the procedure makes use of a procedure known as *displayAt*, an application which causes an image to be displayed at a single location on the screen. The *wallPaper* procedure always accesses the most recent version of the *displayAt* procedure. It always uses the same icon to wallpaper the screen. Figure 2 shows the structure of the store as it would appear after the *wallPaper* application described above has been constructed. The double ellipse surrounding the *displayAt* procedure indicates that the *wallPaper* procedure contains a binding to a location containing the *displayAt* procedure. This enables the *wallPaper* application to make use of the most recent version of *displayAt* contained in that location.

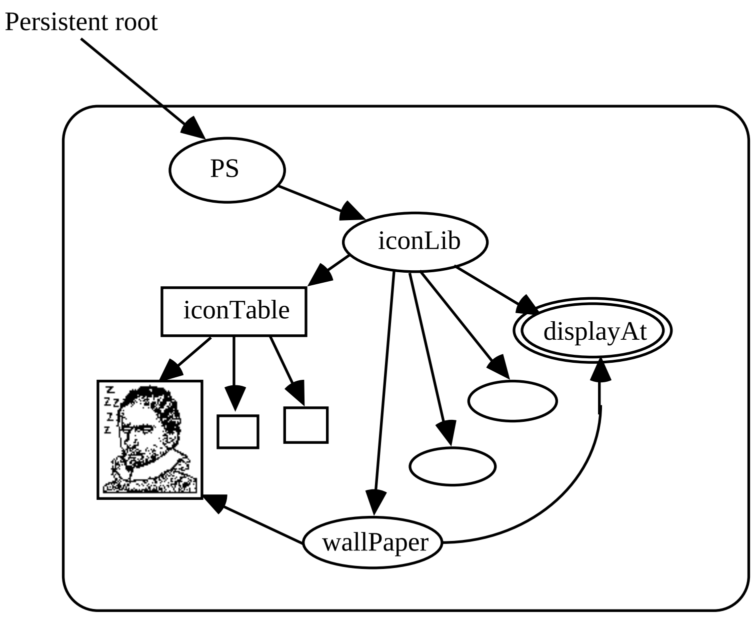


Figure 2: Conceptual view of the completed *wallPaper* application.

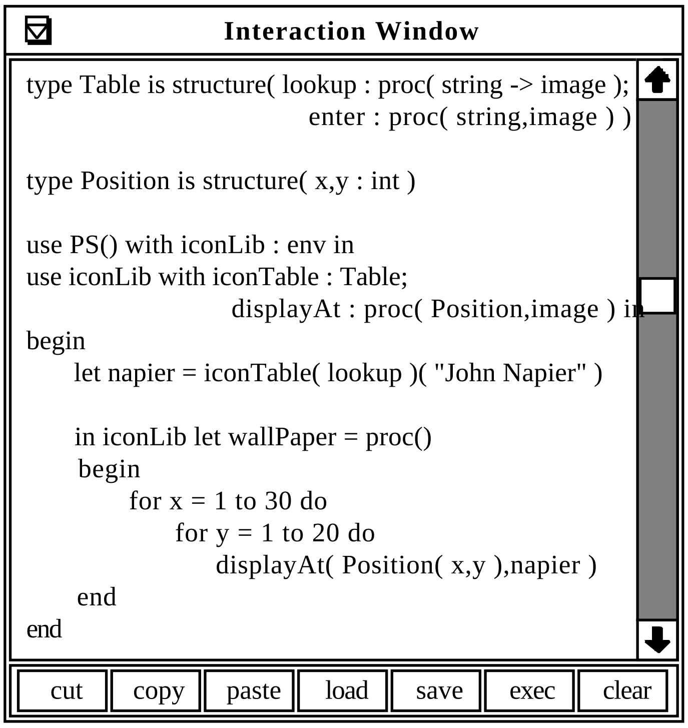
The following sections describe the way in which each of the various programming environment features may be used to implement the *wallPaper* application described above. For brevity, each section only describes the use of either run-time, compile-time or composition-time context in constructing the application, although in general, any combination of these programming styles may be used to construct a single application.

3 Run-time context

3.1 Programming using run-time binding

The first style of programming which shall be described is that in which a program binds to values at run-time. Programs which bind in this fashion must perform a computation which traverses the store to locate the values to be bound. Upon discovering the values in the store, type checking must take place to ensure that the discovered values are of the correct type. This requires the specification of the types in the program to be checked against the types of the values found in the persistent store. For example, in the case of the *wallPaper* application, a computation must be performed to locate the required icon and the *displayAt* procedure.

The programmer is able to construct source code by making use of a text editor known as an interaction window. This editor provides simple text editing features and allows existing source code to be stored or recalled. For example, the application described earlier to "wallpaper" an icon over the screen may be written as shown in the interaction window in Figure 3. The programmer is able to compile and execute this code by selecting the *exec* button.



```
type Table is structure( lookup : proc( string -> image );
                         enter : proc( string,image ) )

type Position is structure( x,y : int )

use PS() with iconLib : env in
use iconLib with iconTable : Table;
                    displayAt : proc( Position,image ) in
begin
    let napier = iconTable( lookup )( "John Napier" )

    in iconLib let wallPaper = proc()
    begin
        for x = 1 to 30 do
            for y = 1 to 20 do
                displayAt( Position( x,y ),napier )
    end
end
```

Figure 3: A window containing an expression to wallpaper an icon.

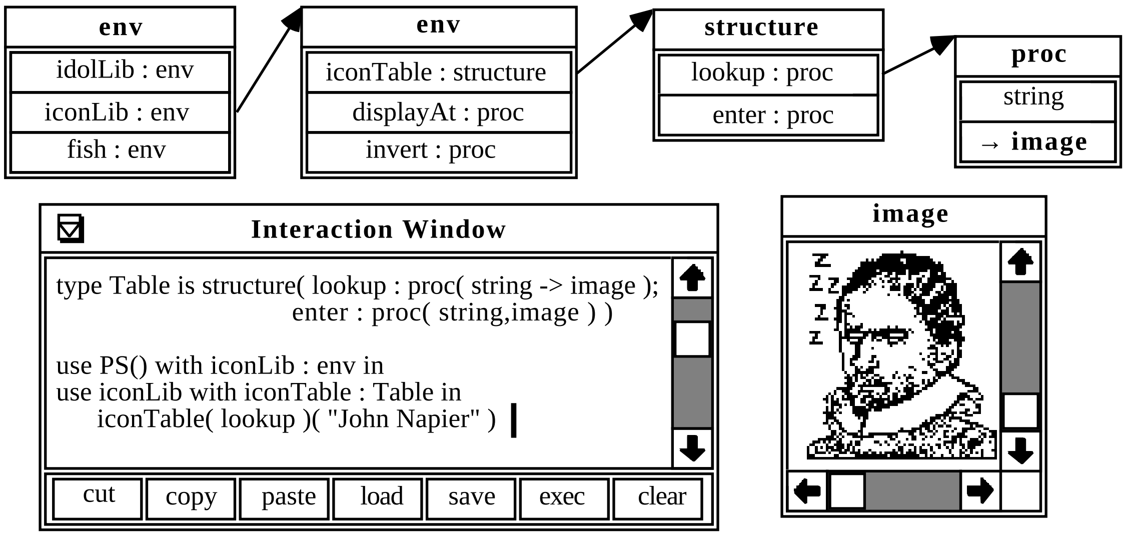
The first three lines of the program declare types; the first two lines define a record type called *Table*. This type contains two fields: *lookup* and *enter*, both of which are procedures. The second type, *Position*, which is a record type, is used to represent a point on the Cartesian plane. Following these declarations are two **use** clauses, which define the names and types of locations expected to be found in the persistent store at run-time. The block following the **use** clauses is statically type checked with respect to these **use** clauses. A once only check is required at run time to ensure that the values found in the store conform to the types specified in the **use** clauses. The bindings to the persistent store that are created when the **use** clauses are executed are bindings to locations. Next, the program declares an identifier called *napier* which is bound to the result of executing the *lookup* procedure from the *Table* structure in the store. The last declaration in the block declares a procedure called *wallPaper*. This declaration is made in the environment denoted by *iconLib* rather than the current scope. The procedure calls the procedure *displayAt* which has been found in the store with a *Position* record and an icon of John Napier as parameters.

The behaviour of the *wallPaper* procedure is such that whenever the procedure is invoked, the most recent version of the *displayAt* procedure is used to display the icon originally assigned to the identifier *napier*. This is due to the fact that the program binds to the location in the persistent store which contains the *displayAt* procedure but binds to the actual value of the image retrieved from the *iconTable*.

A program such as the one described above contains in the source code all of the information necessary to locate values and perform type checking. The process of compilation performs as much type checking as possible. However, correct execution relies on the expected values being present in the store and having the same type as specified by the program.

3.2 Inspecting values

One of the difficulties in constructing programs which bind to values at run-time is that the location and/or types of values in a persistent store may be unknown. For example, a programmer may be aware that a persistent store contains an icon library but may not know how to construct a computation which locates it. To assist the programmer, a tool known as a persistent store browser [3] may be used to inspect the contents of a persistent store in order to discover the location and types of the values in it.



```
type Table is structure( lookup : proc( string -> image );
                         enter : proc( string,image ) )

use PS() with iconLib : env in
use iconLib with iconTable : Table in
     iconTable( lookup )( "John Napier" )
```

Figure 4: Finding values in a persistent store.

In the context of the icon manager example, the programmer may be unaware of the appearance of the icons in the icon table and may not know which icon manipulation procedures are available. A browsing tool may be used to discover the location and types of this data by traversing the store as shown in Figure 4. The browser allows the topology and content of the persistent store to be discovered, thus enabling the programmer to find the information necessary to construct the required program.

The browsing session shown in Figure 4 commences with the traversal of the persistent root displayed on the left hand side of the diagram. The user has selected the field labelled *iconLib* with the mouse resulting in the *iconLib* environment being displayed on the screen. Next, the field labelled *iconTable* was selected causing the table *iconTable* to be displayed. Similarly, the field representing the *lookup* procedure has been selected resulting in a representation of the procedure being displayed on the screen.

We will assume that the programmer knows the names of the icons in the *iconTable*. However, the icons in that table are encapsulated within the closure of the procedures *lookup* and *enter*. In order to examine one of these icons the *lookup* procedure from the table must be invoked with the name of the required icon as a parameter; in general, this requires a program to be written. This may be achieved by entering and executing a small program such as the one shown in the interaction window in Figure 4.

The programmer is required to enter code describing how the values used by the program are located. This is achieved with **use** clauses as described earlier. The result of executing the program is the icon associated with the name "John Napier" and is displayed by the browser as shown on the right hand side of the diagram. The result is displayed because in addition to viewing the contents of the store, the browser may be used to display values returned by expressions entered in an interaction window. In fact the content of the interaction window is treated as a single Napier88 expression which may or may not yield a result upon execution – if an expression yields a result, the result is passed to the browser to be displayed. Having discovered the icons and applications in the persistent store, the programmer is now able to construct a program such as the one shown earlier in Figure 3 which wallpapers the icon onto the screen.

The browser allows the topology of the store to be discovered and allows the programmer to discover information about the location and types of values in the persistent store. However, sometimes in order to manipulate the values and types encountered by the browser a program must be constructed. The code which must be written in order to perform the necessary computation becomes more verbose as the number of values to be bound and the complexity of the path from the root to those values increases.

3.3 Tagging browsed values

The browser supports a mechanism known as "tagging". This mechanism allows the programmer to select a value or location encountered by the browser and to associate a token, or tag, with that value or location. The effect of tagging a value or location is to create a mapping from the tag name to the value or location. Hence tagging represents a way in which tokens may be declared as denoting values or locations.

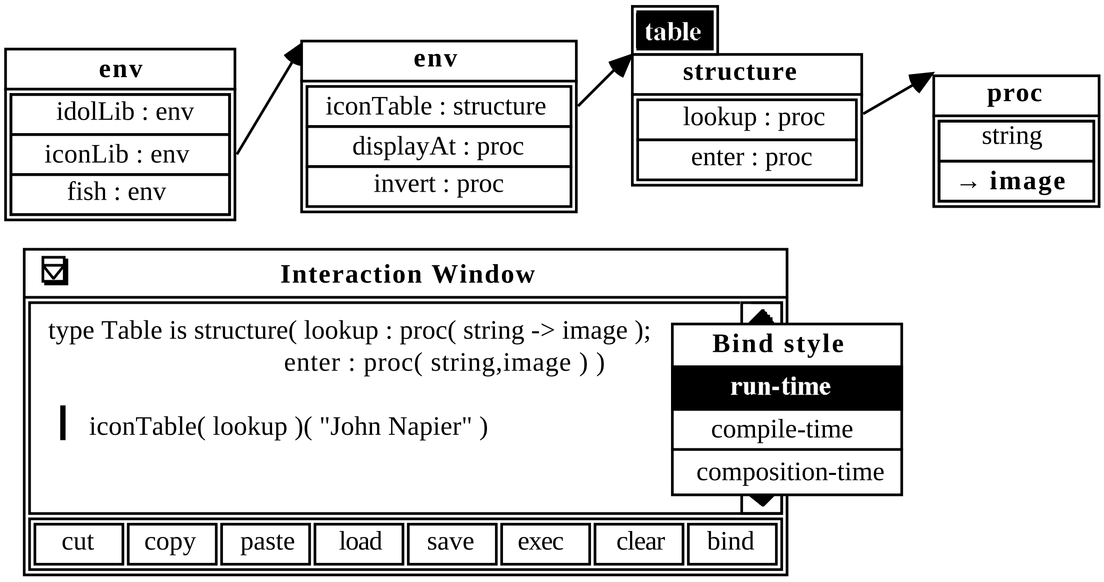
In order to tag a value, the programmer selects the title bar of the window representing that value with the mouse. This causes a dialogue box to be displayed prompting the user for a string to use as the name for the value. The user may enter any string provided it constitutes a valid Napier88 identifier and this string is displayed on the top left hand corner of the tagged value.

The method for tagging a location containing a value is similar to the method used for tagging a value but differs in two distinct ways. The first difference is that instead of selecting the title bar of the value window, the programmer must select a menu entry. The second difference is that when a token has been supplied for the location, it is displayed on the top left hand corner of the value inside a box with a double line border.

In addition to values and locations, the browser also allows types to be traversed. The tagging mechanism may also be used to tag the types encountered by the browser and is the same as the method used to tag values. However, in this case the tag is used to denote a type; in Napier88 there is never any ambiguity over the meaning of this since types are not values.

3.4 Using tags to effect run-time binding

One way to reduce the amount of programmer effort required to produce source code for programs which bind at run-time is to use the browser's tagging mechanism. For example, the two **use** clauses in Figure 4 may be automatically generated using the tagging mechanism as shown in Figure 5.



```
type Table is structure( lookup : proc( string -> image );
                         enter : proc( string,image ) )

  iconTable( lookup )( "John Napier" )
```

Figure 5: Tagging a value encountered by the browser.

Firstly, the programmer has tagged the environment location containing *iconTable* with the name *table*. This is indicated by the double line border surrounding the tag. Next, the programmer has selected the tag using the mouse and has pressed the *bind* button in the interaction window. This has caused the menu entitled *Bind style* to be displayed. At this point, the programmer is required to select which of the three styles of binding the tag is to be resolved into. In this example, selecting the field labelled *run-time* in the *Bind style* menu will cause the two **use** clauses shown earlier to be inserted into the source text at the text cursor's position.

In addition to constructing computations to locate values or locations in the store, the tagging mechanism may be used to construct a textual form for types encountered by the browser. A textual representation of a type may be inserted into source text using the same method as the method for inserting a computation to locate a value or location.

3.5 Reusing programs which bind at run-time

Programs which bind at run-time consist entirely of a textual source code representation. The text editing features provided by the interaction window permit sections of source code to manipulated using facilities such as *cut, copy* and *paste*. As with traditional systems, portions of existing code may be reused.

In addition to source code, values placed in the persistent store by one program may be used by other programs which bind to those values during execution. For example, the *wallPaper* procedure makes use of the *displayAt* procedure defined in another source code segment. Programs which bind at run-time provide the most flexibility in binding to persistent values as they do not require the values to be present at the time the program is compiled or constructed.

4 Compile-time context

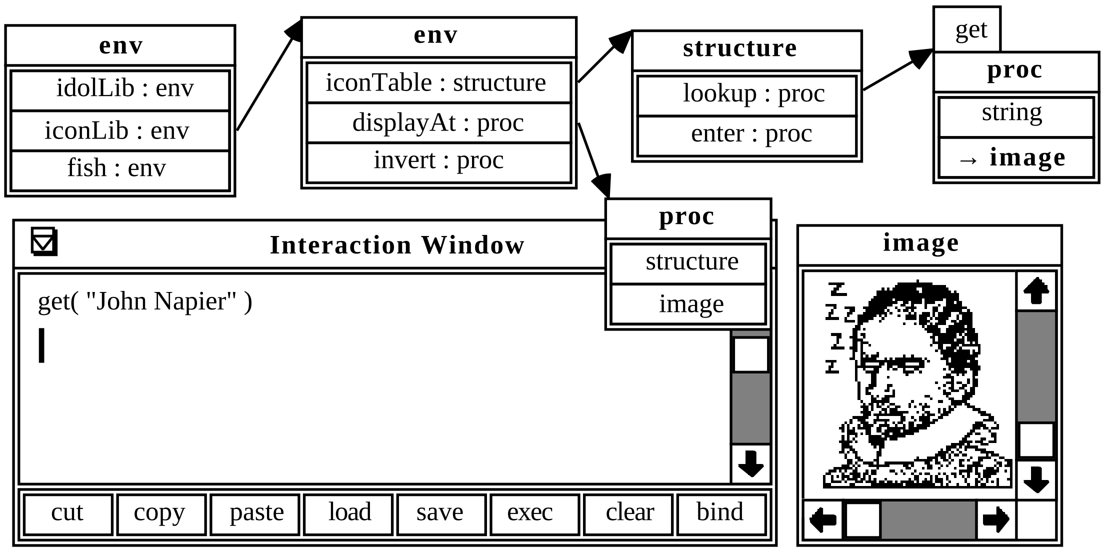
4.1 Programming using compile-time binding

Programs that bind to values at compile-time may be constructed in a similar fashion to programs which bind at run-time: the source code for the program may be entered and manipulated through the interaction window text editor. However, the nature of the source code is different in that source code contains direct references to values in the form of tokens, which are to be resolved at compile-time. When such a program is compiled, the source code of the program is passed to the compiler along with a mapping from tokens to values. The compiler resolves the bindings so that the executable code produced contains references to values and locations. This reduces the verbosity of source code and ensures that referenced values are present at the time the source code is compiled rather than during program execution.

In the programming environment described in this paper there are two kinds of token which may appear in the source code of a program: tokens denoting values or locations and tokens denoting types.

4.1.1 Tokens denoting values or locations

As shown earlier, there is a tendency for the code which must be written in order to locate values to become verbose. Programs which bind to values or locations at compile-time reduce this verbosity by allowing direct references to values in the form of identifiers, or tokens. The mapping from tokens to values and locations generated through tagging is passed to the compiler each time a program is compiled, enabling the compiler to resolve these references.



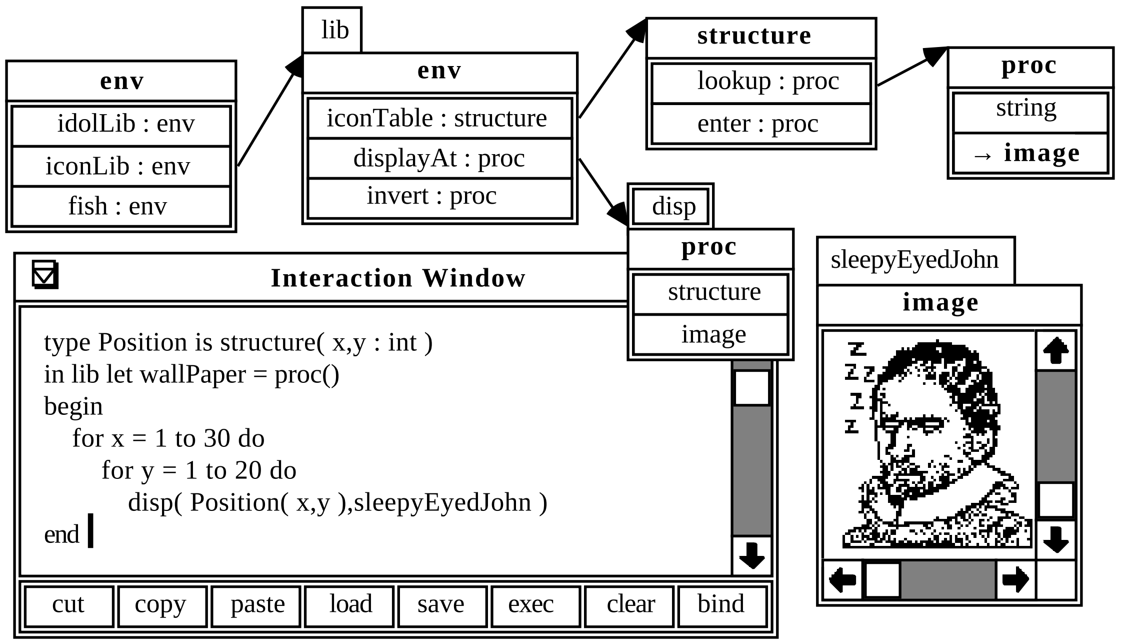
get("John Napier")

Figure 6: A session showing the application of a tagged procedure.

The programmer may create tokens using the tagging mechanism described earlier. In this case, however, the programmer must select the *compile-time* entry of the *Bind style* menu in order insert a compile-time reference to the value into the source text. The resulting reference is simply the name of the tag in the form of an identifier. To illustrate this mechanism, consider the programming environment session in Figure 6 showing a tag on the *lookup* procedure of the icon table *iconTable*. The procedure has been tagged with the name *get* indicating that the token *get* has been declared as a denotation for the procedure.

The programmer may now invoke the *lookup* procedure of the *iconTable* with the string "John Napier" as a parameter as shown in the interaction window of Figure 6. The expression binds to the *lookup* procedure at compile-time through the reference to the token *get*. The binding takes place when the programmer selects the *exec* button, which causes the expression to be compiled and executed. The icon returned by the expression is passed to the browser and displayed as shown on the right hand side of Figure 6.

The second method of tagging allows a location containing a value to be tagged. Using a combination of the two tagging methods, the programmer is able to construct the *wallPaper* procedure described in Section 2. This may be achieved by tagging the necessary values and locations and constructing the source code for the procedure as shown in the interaction window in Figure 7.



```
type Position is structure( x,y : int )
in lib let wallPaper = proc()
begin
   for x = 1 to 30 do
      for y = 1 to 20 do
         disp( Position( x,y ),sleepyEyedJohn )
end
```

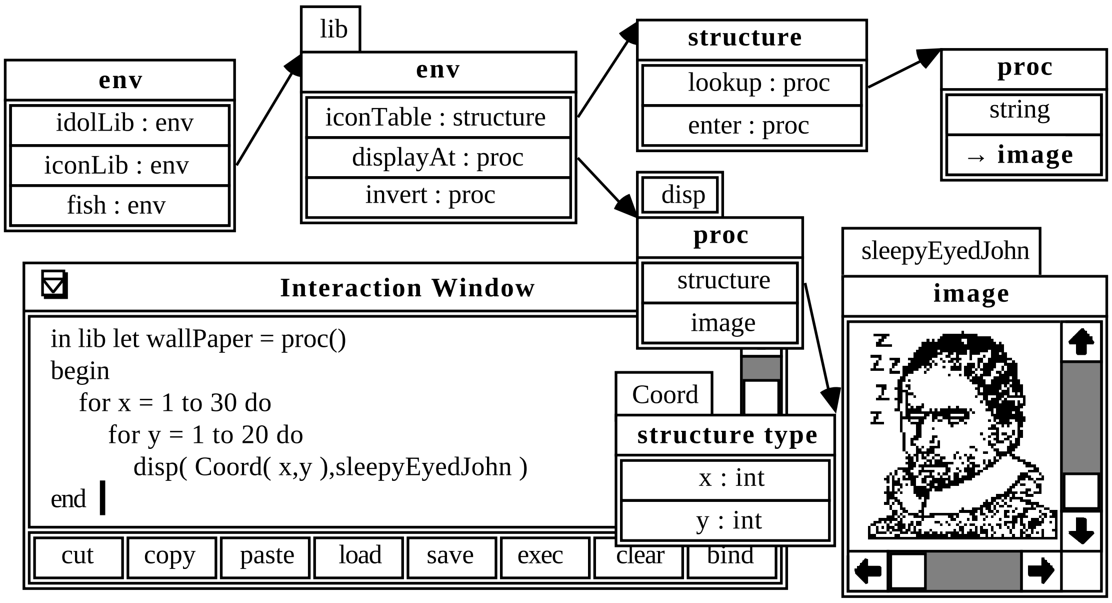
Figure 7: Constructing the *wallPaper* procedure using compile-time binding.

The procedure *displayAt* has been tagged with the name *disp* in a box with double line borders, indicating that the binding is to the location containing the *displayAt* procedure rather than its value. Next, the environment *iconLib* and the icon returned by the expression in Figure 6 have been tagged with the names *lib* and *sleepyEyedJohn* respectively. The plain boxes indicate that these tokens represent bindings to actual values rather than locations containing values. Lastly, the code entered in the interaction window declares the type *Position* and the procedure *wallPaper*. The program places the *wallPaper* procedure in the *iconLib* environment by declaring the procedure in the environment denoted by the token *lib*.

4.1.2 Tagging types

In the above example the program must declare the type *Position* because it is used to construct a value supplied as a parameter to the procedure *disp*. However, as the complexity and number of type declarations required by a program increases, the source code once again becomes verbose. Furthermore, an increasing proportion of the time taken by a programmer to construct a program is spent entering these type declarations.

By tagging the appropriate type and inserting a compile-time binding to that type, the code shown in Figure 7 may be rewritten as shown in Figure 8.



```
in lib let wallPaper = proc()
begin
    for x = 1 to 30 do
        for y = 1 to 20 do
            disp( Coord( x,y ),sleepyEyedJohn )
end
```

Figure 8: Using a tagged type in a program.

The programmer has selected the first entry of the *displayAt* procedure and this has caused the type of the first formal parameter of the procedure to be displayed. This type has been tagged with the name *Coord* and a compile-time reference to the type has been inserted in the source code in place of the first actual parameter supplied to the *disp* procedure. The declaration of the type *Position* has been omitted.

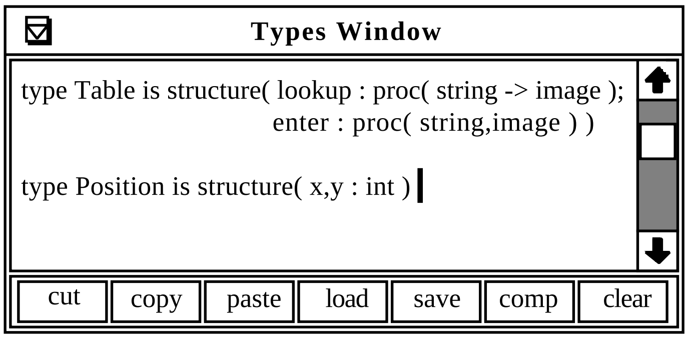
Thus, using compile-time references to tagged types, the verbosity of the code may be further reduced. Moreover, the process of compilation becomes more efficient since the need to recompile type declarations is reduced, or sometimes removed.

4.1.3 Tokens denoting types

An instance of a required type may not currently exist in the persistent store. Therefore some alternative method for creating tokens that represent types is required. The Napier88 programming environment supports a structure known as a type environment: a mapping from names to types which may be passed to the compiler to permit the use of types that have not been declared in the source program. This is achieved by the compiler resolving references to types contained in the type environment and used, but not declared, in the source program. The compiler is also used to create type environments, a string containing type declarations is passed to the compiler and a type environment is returned. This presents an alternative means by which tokens denoting types may be created and stored.

In the programming environment described in this paper, the following interface is provided to ease the creation of type environments. Types may be declared in a separate window known as the types window; this window is similar to the interaction window in that it consists of a text editor. However, rather than allowing arbitrary Napier88 programs to be entered, the types window only permits the declaration of types. Figure 9 shows the appearance of the type window when the types *Table* and *Position* are declared.

When the programmer presses the *comp* button, the text in the types window is compiled and a type environment is created. This type environment is implicitly part of the compilation environment of the interaction windows provided by the user interface. Therefore once a type environment is created, the programs do not need to declare the types being used. In this example this means that the types *Table* and *Position* may be used without declaring them in the source code.



```
type Table is structure( lookup : proc( string -> image );
                        enter : proc( string,image ) )

type Position is structure( x,y : int )
```

Figure 9: Declaring types in the type window.

In the example shown in this paper, we make two assumptions for simplicity, they are:

1. there is only one type environment in existence, and
2. the type environment implicitly forms part of the compilation context for interaction windows.

In practice, a programmer needs to use many different type environments – each tailored to the task in hand. Therefore the system needs to permit more than one type environment to exist and provide some mechanism to associate an arbitrary collection of type environments with an interaction window.

In addition to removing the need to recompile type declarations, type environments provide a means by which different programs may share type declarations as well as reducing the verbosity of programs which use a large number of types.

4.2 Editing compile-time context programs

The programmer is able to manipulate the source text of a program which binds at compile-time in the same ways as described in the previous section. However, by changing the mapping from tokens to values or types, the programmer is able to construct different applications using the same source text. For example, the programmer may bind the token *sleepyEyedJohn* shown in Figure 7 to a different icon in order to change the semantics of the application without having to alter the source text. More generally, the same source code may be used with different token mappings in order to produce a sequence of applications which vary depending on the values to which they are bound at compile-time. In this way, the nature of programs written using compile-time binding changes in comparison to programs which use only run-time binding.

5 Composition-time context

Binding to values at program composition-time is supported by *hyper-programming* [5]. A hyper-program is a source code representation that contains embedded bindings. This section outlines the main differences between this style of programming and the style described in the previous section.

5.1 Hyper-program source representations

In a hyper-program, the bindings embedded in a hyper-program are an integral part of the program. This contrasts with the compilation-time binding style where the source program and the mapping from tokens to values are distinct entities that are presented to the compiler separately. The physical realisation of that mapping depends on the interface provided to the programmer: with the tagging mechanism described earlier, the mapping is implicit in the tags that are present at the time of compilation. As described earlier, the source code of programs which make use of compile-time binding may be compiled with different mappings (by tagging different values) to give different executable programs. In a hyper-program, however, the bindings from tokens to values do not need to be resolved by the compiler as the resolution takes place earlier, at the time the program is constructed.

To provide flexibility, a hyper-programming system should support all three styles of binding and allow the programmer to choose the appropriate style for each application component. This would allow a source program to contain tokens that are already bound to values, tokens that will be matched with values at compilation-time, and expressions that will be evaluated to give values at run-time. As previously described, for brevity the example in the next section shows only composition-time binding.

5.2 Constructing hyper-programs

5.2.1 Method of construction

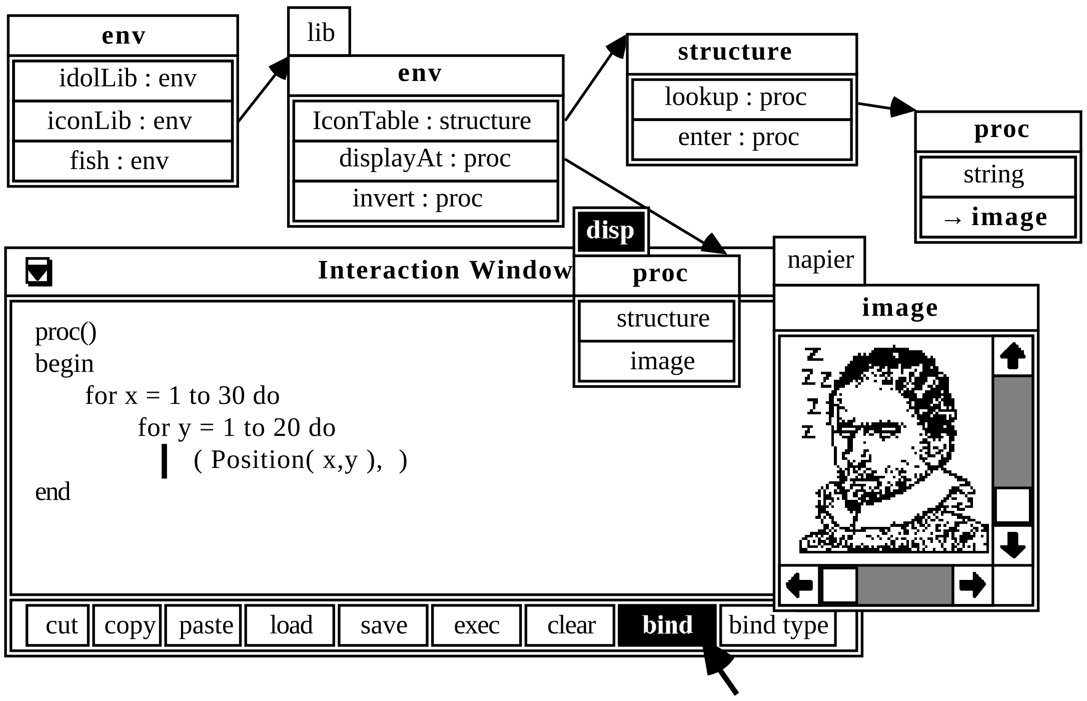
A hyper-program is constructed in a similar way to the construction of programs that contain compilation-time bindings. The programmer types textual code into an interaction window and uses browsing tools to navigate the persistent store to locate values to be bound into the program. The difference is in how the binding is effected; with compilation-time binding the programmer attaches a tag or token name to each value required and enters the corresponding token name at the appropriate point in the source text. The tokens in the code and the tokens attached to the value representations are matched by the compiler. To achieve composition-time binding, the programmer first tags the desired value and then selects the *composition-time* entry of the *Bind style* menu described earlier to bind that value into the program. The system inserts a button into the text to act as a place-holder and to allow the programmer to later examine the bound value. When that button is pressed subsequently, a representation of the value is displayed by the browsing tool.

The system allows the programmer to insert bindings to values themselves or to environment locations. As described earlier, the way in which a tag is effected will determine whether a binding is to a location or value.

It is also possible to bind a type into a program using the same method as the method for binding values into a program. This may also be achieved by selecting a type in the type environment window and pressing the *bind type* button. This mechanism reduces the number of type definitions that the programmer has to enter. As with values, a bound type can be examined by pressing the associated button

5.2.2 An example

This section illustrates how the *wallPaper* application described earlier may be constructed in a hyper-programming system. The first step is, as before, to tag the procedure that performs a look-up on the icon table and to execute some code to invoke the procedure in order to obtain the icon for John Napier. This process was illustrated earlier in Figure 6 of Section 4. Next, the programmer enters the textual part of the application, leaving gaps where values are to be bound into the code as shown in Figure 10. Note that the source code of the expression contained in the interaction window represents the declaration of a procedure – hence the result of evaluating the expression is the value of the procedure itself and not the execution of the body of the procedure.



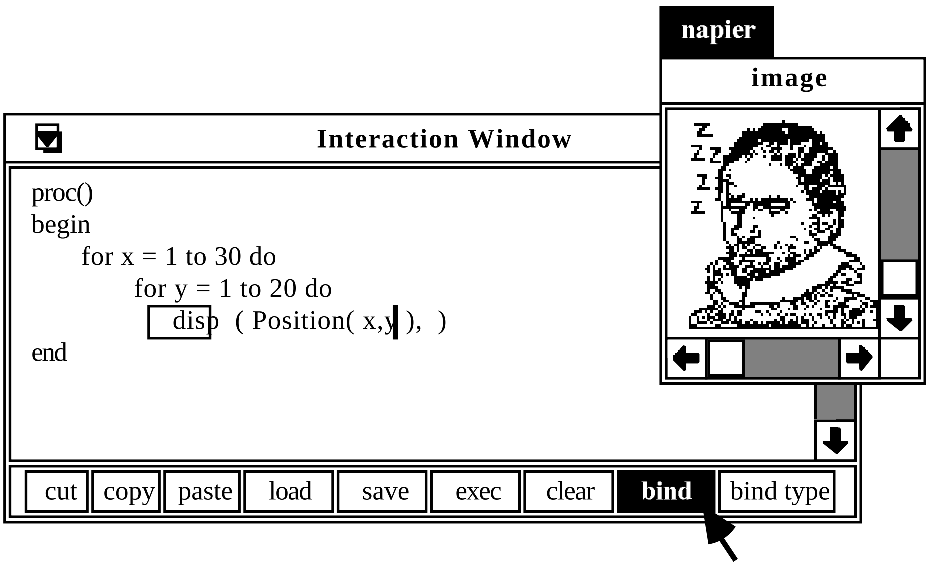
```
proc()
begin
     for x = 1 to 30 do
          for y = 1 to 20 do
               ( Position( x,y ), )
end
```

Figure 10: Binding a location into source code.

To bind the location of the *displayAt* procedure into the hyper-program the programmer uses the browsing tool to locate and tag a representation of the location and then presses the *bind* button in the interaction window. When the *Bind style* menu is displayed, the programmer selects the *composition-time* entry in order to insert a binding at the current text position. This is illustrated in Figure 10.

This inserts a button into the text to denote the binding to the selected environment location. The name displayed in the light button is the name of the tag. However, although a name for the button is not essential, we are accustomed to names in our programs so this probably makes the program easier to read.

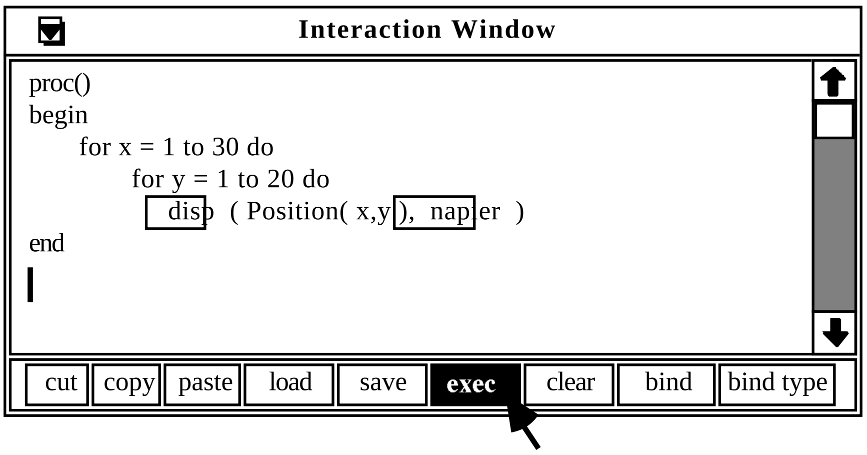
A similar procedure is followed to bind the icon into the hyper-program but this time the programmer must tag the value of the icon in order to bind to the icon itself rather than its location. Note that in this case there is no corresponding location which may be bound to. The appearance of the interaction window at this stage is shown in Figure 11.



```
proc()
begin
    for x = 1 to 30 do
        for y = 1 to 20 do
          disp ( Position( x,y ), )
end
```

Figure 11: A light button in the source text indicating a binding to the location of the *displayAt* procedure.

The completed program is shown in Figure 12. To execute the program the programmer presses the *exec* button which will cause a representation of the resulting procedure to be displayed.

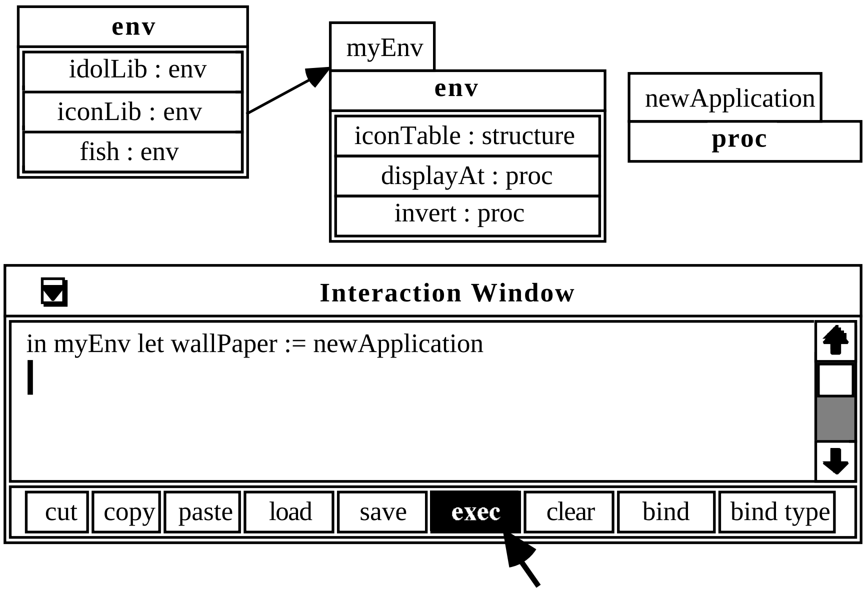


```
proc()
begin
    for x = 1 to 30 do
        for y = 1 to 20 do
          disp ( Position( x,y ), napier )
end
```

Figure 12: A complete hyper-program.

To make the procedure persist the programmer creates a binding to it in the environment *iconLib.* For example, one way to achieve this is by tagging the representations of the environment and the new procedure and executing some code to create the binding as shown in Figure 13.

The new application is now complete and may be accessed from the environment *iconLib.* In order to prevent the icon from being removed or corrupted by the actions of other programs, the value of the icon is bound into the application. Assignments to the environment location containing the *displayAt* procedure, however, will affect the *wallPaper* application: it will always use the procedure assigned to that location at the time the application is executed. The application's access to the *displayAt* procedure does not depend on the path that the programmer initially followed through the store when binding it into the hyper-program. For example, the binding to *displayAt* may be dropped from the *iconLib* environment without affecting the application or the hyper-program that represents it.



```
in myEnv let wallPaper := newApplication
```

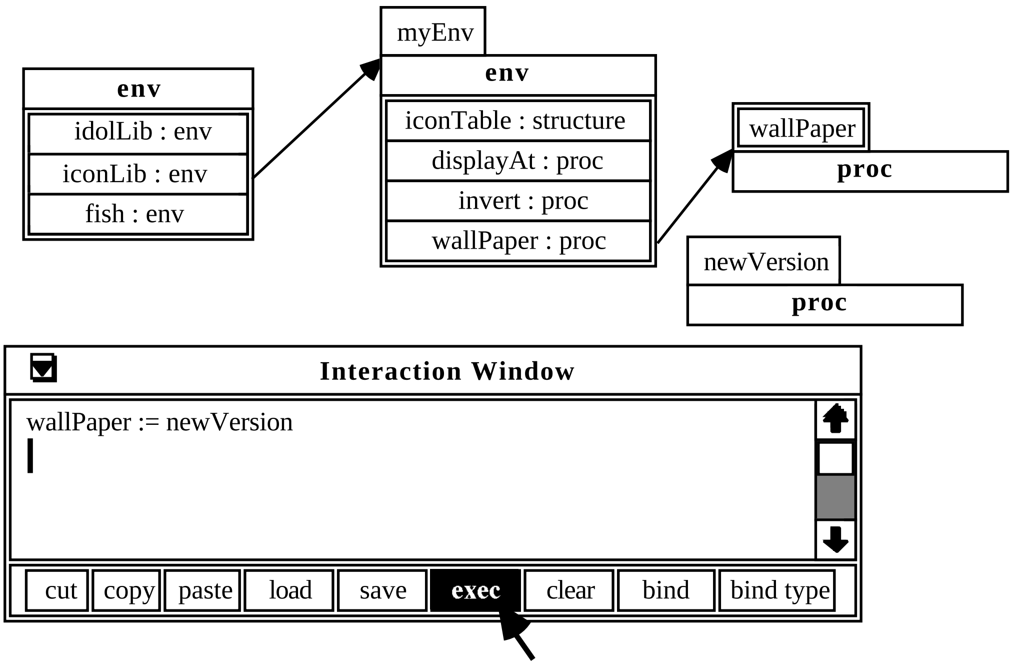
Figure 13: Placing the *wallPaper* application into the icon library.

5.3 Editing hyper-programs

The programmer can later refine the implementation of the application whilst leaving the bindings to its components intact. For example, the **for** loops may be changed so that the icon is drawn only around the edge of the screen rather than over the whole screen. To achieve this, the programmer selects the representation of the procedure and directs the system to supply its source code. This results in the display of a new interaction window containing a copy of the original hyper-program. Although a copy, it contains bindings to the same values and locations as the original source code. The programmer then edits the text of the new hyper-program and presses the *exec* button. If compilation is successful the representation of a new procedure is displayed by the browser.

This new procedure has different behaviour from the original application but contains the same bindings. Finally the programmer tags the new procedure and the location *wallPaper* in *iconLib* and executes some code to assign the procedure to the location as shown in Figure 14.

The technique of refining implementation whilst retaining state may be used in other cases. For example the programmer may discover that there is a bug in the implementation of the procedures that operate over the table of icons. By editing copies of the hyper-program source the programmer may correct the error and install a new version without losing the existing contents of the table. More generally, this provides a mechanism for repairing abstract data types without throwing away their state.



```
wallPaper := newVersion
```

Figure 14: Updating the value in the location of the *wallPaper* procedure.

6 Current status and future work

To date two experimental systems [1,4] have been constructed to allow the exploration of the ideas described in this paper. Both the prototypes support the compile-time binding paradigm described in the paper. In addition, a system which permits composition-time binding has been constructed at the University of St. Andrews. Our plans for the future involve integrating the best features of these prototypes and we expect these facilities to manifest themselves in future releases of the Napier88 programming environment. A final field of research which remains untapped is the program development environment required to support this new kind of programming. This will be the subject of future research.

7 Conclusions

In this paper we have described a mechanism for program construction and manipulation using gesture. Such a paradigm differs greatly from conventional programs which consist of denotations which describe an algorithm and some values which may be constructed at run-time. The mechanisms described in this paper permit this conventional style of programming but augment it with the ability to describe values which exist at compilation- or construction-time. The new programming style allows programs to be constructed which are shorter than those previously expressible using persistent languages like Napier88. This brevity is not achieved without cost, the programs are more tightly bound than other Napier88 programs. We do not imagine that such a mechanism will be appropriate for all programming tasks. Instead, we assert that the new style of programming which emerges from this ability will augment techniques already at the disposal of the programmer.

Acknowledgments

This work is supported in part by ESPRIT III Basic Research Action 6309 – FIDE 2 and SERC grant GR/F 02953. Richard Connor is supported by SERC Post Doctoral Fellowship B/91/RFH/9078.

We would also like to thank the Defence Science and Technology Organisation of Australia for their assistance through the PIPE project, The University of Adelaide through its University Grants Scheme and the Australian Research Council.

References

[1] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A. and Morrison, R., "Programmer's Guide to the Napier88 Standard Library (Edition 2)", Reference Manual, 1991, University of St. Andrews.

[2] Dearle, A., "Environments: A Flexible Binding Mechanism to Support System Evolution", in Proceedings of the 22nd Hawaii International Conference on System Sciences, 1989.

[3] Dearle, A. and Brown, A.L., "Safe Browsing in a Strongly Typed Persistent Environment", The Computer Journal, 1988. Vol. 31, No. 6: pp. 540-545.

[4] Farkas, A.M., "ABERDEEN: A Browser allowing intERactive DEclarations and Expressions in Napier88", Honours Report, 1991, University of Adelaide.

[5] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. and Morrison, R., "Persistent Hyper-Programs", in Proceedings of the Fifth International Workshop on Persistent Object Systems, San Miniato, Italy, 1992.

[6] Morrison, R., Brown, A.L., Connor, R. and Dearle, A., "The Napier88 Reference Manual", 1989, University of St. Andrews.