This paper should be referenced as:

Kirby, G.N.C. & Dearle, A. “An Adaptive Graphical Browser for Napier88”. University of St Andrews Technical Report CS/90/16 (1990).

An Adaptive Graphical Browser for Napier88

G.N.C. Kirby and A. Dearle†

University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland.
†University of Adelaide, GPO Box 498, Adelaide, South Australia 5001, Australia.

Abstract

A persistent store, such as that found in Napier88, allows data objects to outlive the execution of a particular program. Using the persistent store entails binding some of the objects in the store to a use in a program invocation. Since the store may evolve incrementally and through the actions of many users, a need arises to discover the names, types and values of objects within the persistent store to effect this binding.

This need is met by a browser: a polymorphic program which takes as its parameter any object, and displays some representation of that object. We describe the use and implementation of a graphical browser for Napier88 objects, itself written in Napier88. The browser is adaptive in that it learns incrementally about the types of the objects it encounters.

1 Introduction

A persistent store, such as that found in Napier88 [MBC89], allows data objects to outlive the execution of a particular program. Using the persistent store entails binding some of the objects in the store to a use in a program invocation. Since the store may evolve incrementally and through the actions of many users, a need arises to discover the names, types and values of objects within the persistent store to effect this binding.

This need is met by a browser: a polymorphic program which takes as its parameter any object, and displays some representation of that object. A graphical representation is often helpful in viewing complex objects, which in Napier88 may be structures, variants, abstract data types, procedures, environments, bitmaps and line drawings, as well as the usual scalar types. We will describe the use and implementation of a graphical browser for Napier88 objects, itself written in Napier88. The design of this browser is based on an existing browser [DB88],[Dea88] for the language PS-algol [PS87].

The current version of the Napier88 browser differs from the previous version in that it gives a graphical representation of the links between objects. This makes it easier for the user to comprehend the descriptions of complex data structures. The browser implements most of the facilities described in [DCK89]. The previous version of the browser is described in an earlier report [KD90].

Section 2 describes in general terms the design of the browser; in Section 3 we describe how the user can make and invoke a browser; Section 4 describes the implementation of the browser in more detail; Section 5 describes an alternative implementation method with which we have experimented; finally in Section 6 we suggest ways in which the browser system could be improved.

2 Design of the Browser

A requirement of the browser is that it should operate on objects of any type. Furthermore, it should operate automatically for all types, even those which it has never encountered previously. That is, it should not rely on the programmer, on defining a new type, providing a method of displaying objects of that type. Such a scheme is found in Smalltalk-80 [GR83], and is unsafe in that the programmer could code any arbitrary action in the method used by an object to respond to the **print** message. Our browser should itself decide on the appropriate action when presented with an object of a novel type.

There are potentially an infinite number of legal Napier88 types, therefore as browser designers we cannot hope to specify individually the action to be taken for each possible type. However, there are only a small number of type **constructors**, and all the types generated by a particular constructor can be treated in a similar way. For example, this browser displays all structure types as menus with entries for each field, while all vector types are shown as menus which allow the user to inspect the bounds and the elements of the vector.

Procedures for displaying the eleven base types are built into the browser. When presented with an object of some other type, i.e. a constructed type, it builds the text of a new Napier88 procedure to display objects of that type. Thus for each constructor, the browser has generic knowledge of how to display the objects whose types can be produced with that constructor. The browser then compiles the text, using the Napier88 compiler which is available as a procedure in the persistent store. The resulting procedure is then used to display the object.

This strategy would be very expensive if a new procedure had to be compiled every time the browser encountered anything other than a base type. Instead, the browser stores each procedure

that it compiles. If other objects of the same type are presented to it, that display procedure is recalled and used without any further compilation. In this way the browser is adaptive: the knowledge which it acquires is retained. As the Napier88 store is persistent, the procedure building process is only ever necessary on the first encounter with a particular type.

The browser is implemented as a procedure which can take any Napier88 value as its parameter. To allow this its argument is coerced to type *any* before being passed to it. The type of the browser procedure is thus *proc(any)*.

3 Using the Browser

This section describes how to configure an instance of the browser using procedures from the persistent store, how to use it to examine objects, and the ways in which the various groups of types are displayed.

3.1 Making a Browser

The browser uses the WIN window management system [CDK89] to provide a graphical user interface. The browser system provides a generating procedure, which takes as its parameter an instance of a WIN window manager, and returns a browser, a procedure of type *proc(any)*. This can then be called with some value injected into type *any* as its argument. Window managers are described in the WIN Programmer's Manual [CDK90].

The browser generator is called *travGen*, standing for *traverser generator*, and is of type

travGen : **proc**(WindowManager → **proc**(**any**))

This procedure is located in the environment *browser2*, in the environment *Lib*, in the root environment. Its use is illustrated in the Napier88 code shown in Figure 1.

```
use PS() with Lib : env in
use Lib with browser2 : env in
use browser2 with travGen : proc( WindowManager → proc( any ) ) in
begin
    let windowManager = …                          !*** Code to make a window manager.
    let trav = travGen( windowManager )            !*** trav is of type proc( any )

    trav( any( "hello mum" ) )
    trav( any( struct( a = 7 ; b = false ) ) )
    trav( any( a complex data structure ) )
end
```

Figure 1: Configuring and Calling a Browser

3.2 Browser User Interface

The browser is invoked with code such as that shown in Figure 1. The object passed to the browser is then shown in a window which is displayed on the screen. The appearance of the window depends on the type of the object.

Complex objects such as structures, variants and vectors are displayed as menus. The user can discover the sub-components of these objects by selecting the appropriate menu entry with the mouse; this has the effect of calling the browser again with the component object as parameter. The windows for many different objects may be visible at one time, and the browser shows the links between complex objects by drawing arrows. These connections are described in *Section 3.2.2 Exploring Links*.

3.2.1 Displaying Objects

Individual objects are displayed in various ways depending on their type:

scalars Integers, reals, strings, booleans, and pixels are displayed in a text output window.

e.g. trav(**any**(3.2))
trav(**any**("hello mum"))

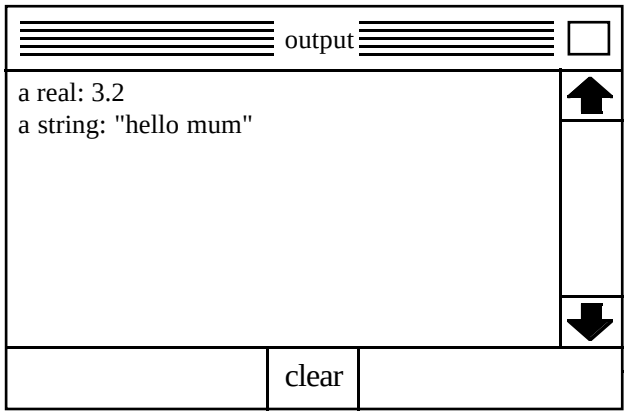

Figure 2: Displaying Scalars

If the output window is already displayed when a scalar is encountered, but obscured by other windows, it is brought to the front so that is fully visible. The text in the output window can be scrolled using the arrow buttons or cleared with the ‘clear’ button. The user may resize the window by clicking mouse button 1 in the box on the top border and then dragging a corner or side in or out. The window can be moved by dragging the top border using mouse button 2.

structures These are displayed as menu windows with an entry for each field of the structure. Each entry gives the name of the field, and its type if it is a base type or vector, or one of ‘STRUCT’, ‘VARIANT’, ‘PROC’ or ‘ABSTYPE’ if it is a constructed type. It is not possible for the browser to display the name of the structure type, as this information is not carried with structure objects at run time. Indeed, Napier88 allows the creation of anonymous structures without first defining their type.

e.g. **type** X **is structure**(a : **int** ; b : **structure**(…) ; c : **proc**(…), d : **env**)
*!*** Declare objects i,j,k,l with the appropriate types.*
trav(**any**(X(i,j,k,l)))

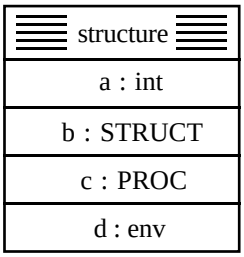

Figure 3: Displaying a Structure

When an entry is selected using mouse button 1 the browser traverses the value of the appropriate field, which may in turn cause a new window to be displayed.

variants These are displayed in the same way as structures, except that the entry for the branch of the variant which is actually present is shown in bold type. When this entry is selected the browser traverses the value of the branch; selecting other entries has no effect.

e.g. **type** X **is variant**(a : **variant**(…) ; b : **abstype**(…) ; c : **image**)
*!*** Declare i of type image.*
trav(**any**(X(c : i)))

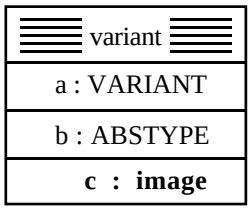

Figure 4: Displaying a Variant

vectors The menu shown in Figure 5 is displayed.

e.g. trav(**any**(vector @1 of ["the","quick","brown","fox"]))

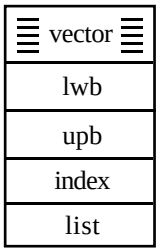

Figure 5: Displaying a Vector

Selecting ‘lwb’ or ‘upb’ causes the lower bound or upper bound of the vector, respectively, to be displayed in the text output window. The user can examine an individual element of the vector by selecting ‘index’ and entering the index of the element in the dialogue window which appears. Selecting the ‘list’ entry causes the

browser to traverse all of the elements of the vector consecutively, in ascending order.

environments These are displayed in the same way as structures, each menu entry showing the name and type of a binding in the environment. If the environment contains more than six bindings a scrolling menu is used, so as to keep the display compact.

```
e.g. let x = environment()
     in x let a = variant( … )
     in x let b = abstype( … )
     in x let c = image 10 by 10 of on
     in x let d = 1
     in x let e = "abcd"
     in x let f = true
     in x let g = 4.2
     trav( any( x ) )
```

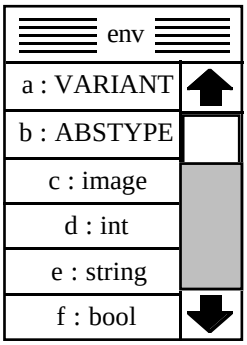

Figure 6: Displaying an Environment

images These are displayed in a window such as that shown in Figure 7.

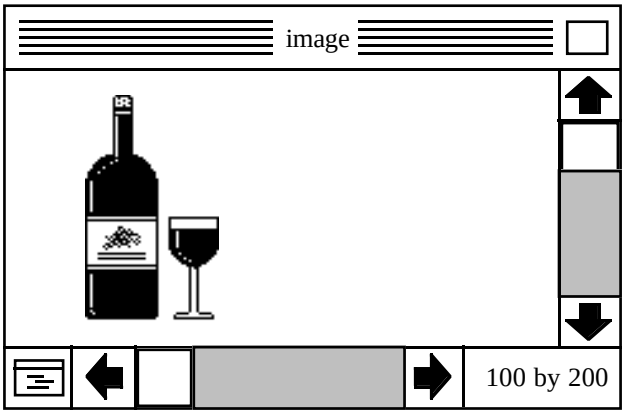

Figure 7: Displaying an Image

The scroll bars may be used to scroll the image so as to reveal any parts not visible. The size of the image in pixels is shown in the bottom right corner of the window. The user may resize the window by clicking mouse button 1 in the box on the top border and then dragging a corner or side in or out. However, the window cannot be made smaller than the size necessary to show the scroll bars.

Normally, successive images are displayed in the same window, so when an image is shown any previous image is erased. This is to maintain consistency with the display of scalar values, where all descriptions are output to the same window. Another motivation for this scheme is that the user may wish to scan quickly through a vector of images, such as might be stored in a font description or for animation purposes. Creating a new window for each image would be too slow for this.

However, in some cases the user will want to be able to see more than one image simultaneously, and to allow this the display window may be duplicated. When the button in the bottom left corner of the window, showing a window icon, is pressed, a copy of the window is created and displayed. The new window contains the same image as the original window. If the browser subsequently encounters another image, the image in the original window will be overwritten but the one in the new window will remain unchanged. This process can be repeated any number of times; new images are always shown in the original window. Only the original window can be duplicated; the copies of it do not contain the duplication button.

pictures These are displayed in a window similar to that used for images. When the picture is first drawn, it is scaled so that it fits completely into the display area. Resizing the window enlarges or reduces the display area but does not alter the scale of the picture, so that more or less of the picture will become visible. The scroll bars can be used to pan the display area to a different region of the picture.

The window also contains two arrow buttons on the left hand side which can be used to enlarge or reduce the scale at which the picture is drawn. This makes it possible to zoom in on a region of interest or to move back to view the picture as a whole.

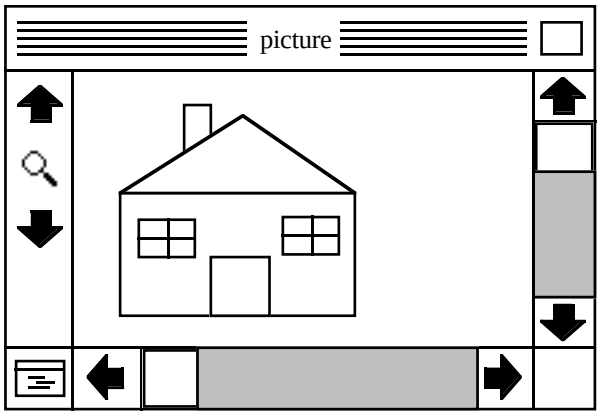

Figure 8: Displaying a Picture

The mechanism for duplicating the image display window is also used with the picture display window.

procedures The first implementation of the browser provided no useful information on encountering a procedure, simply writing ‘a proc’ to the text window. It would be preferable to show the source text of the procedure, and to allow the user to browse the objects in its state i.e. the objects referred to but not defined in the text.

This is possible only where the PAIL (an abstract syntax description of the procedure code [Dea87]) has been included in the procedure closure; this depends on the version of the Napier88 compiler used to construct the procedure. The browser inspects each procedure to determine whether the PAIL is present. If not it outputs ‘a proc with no PAIL’ to the text window.

If the PAIL is present the browser displays a window showing a textual representation of the source, any type definitions necessary, and a menu with free variables. An example is shown in Figure 9.

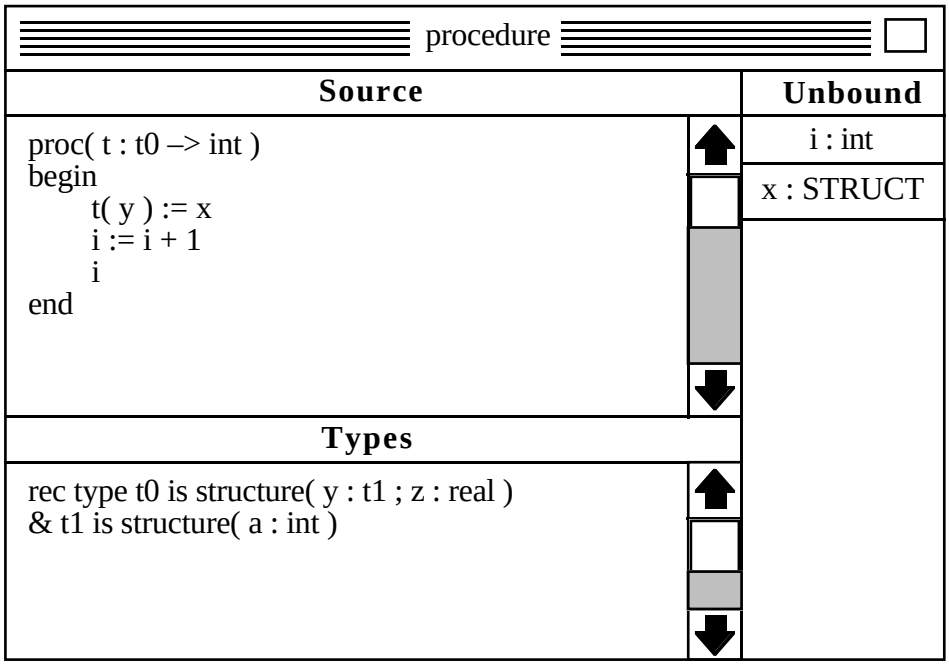

Figure 9: Displaying a Procedure

The definitions of the non-base types used in the procedure are displayed in a sub-window below the code itself. It is often not possible to reproduce the name given to a type by the programmer at the time of writing the procedure. In that case the types are called *t0, t1, t2* and so on.

The values of the free variables can be browsed by selecting the appropriate entries in the menu on the right.

anys Although constructed objects may contain elements of type *any*, no special action is required by the browser for traversing those elements. This is because the action of coercing a value to type *any* is idempotent, that is, performing it more than once has the same effect as performing it once only. When an object of type *any* is itself coerced to *any* before passing it to the browser, no further change takes place, and the browser simply traverses the value which was originally injected.

others At present no information about abstract data types or files is displayed. The browser simply outputs ‘an abstype’ or ‘a file’ to the text window.

3.2.2 Exploring Links

As has been described, the user can examine the components within an object by selecting entries from the object's menu. The menus for both the original object and its components may be visible on the screen simultaneously. To aid the user in remembering the relationships between the objects, the browser draws directed links between the menus. For example, Figure 10 shows the menu for a structure with two fields, one containing an integer, and the other pointing to some other structure.

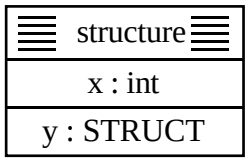

Figure 10: A Single Object

When the user selects the second entry the structure pointed at is displayed, and an arrow drawn to it from the menu entry. This is shown in Figure 11.

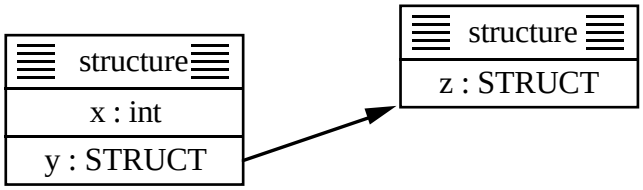

Figure 11: An Object Linked to a Component

When the user examines the field *x* no arrow is drawn, as all scalar values are displayed in the same text window. The same is true for fields of type *image* and *pic*. However, arrows are drawn to all structures, variants, vectors, environments and procedures.

Because data structures may be cyclic, it is possible that a component of an object might be already displayed when that object was first drawn. However, links are drawn only when explicitly requested. This means that the link from the object to the component would not be drawn until the user selected the menu entry for that component.

The display of links becomes more useful when browsing data structures of greater complexity, such as that shown in Figure 12:

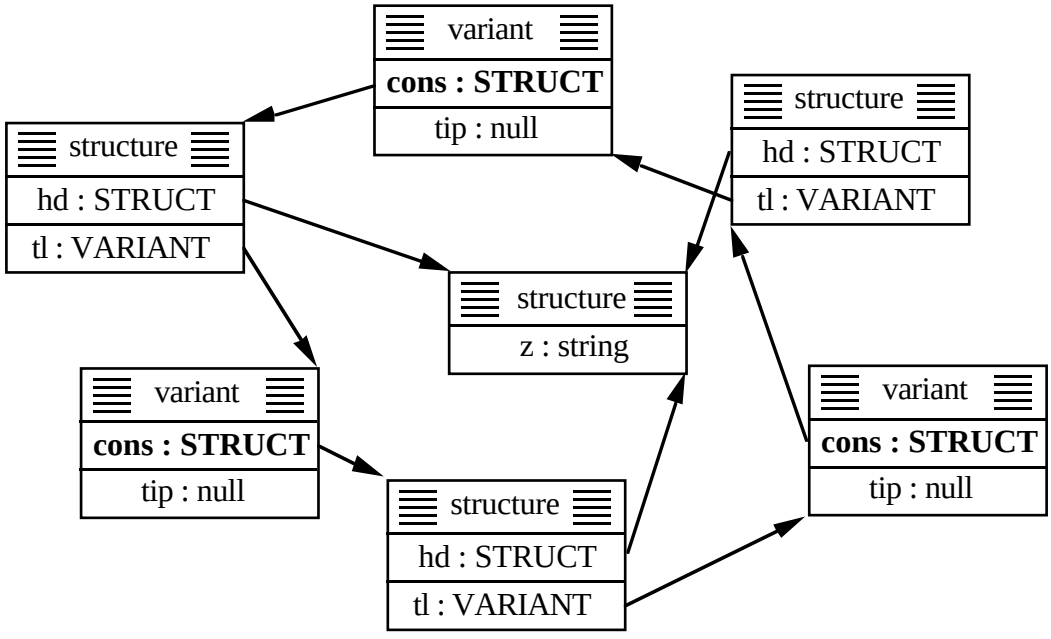

Figure 12: A Circular Linked List with a Common Element

The arrows in the browser display make it easy to see that the list has a circular structure and that all three list cells share the same element. Note that this example shows the display as a user might arrange it: the browser would initially distribute the objects in a linear pattern.

Figure 13 shows part of the environment hierarchy in the persistent store. Again, the links emphasise the relationships between the environments.

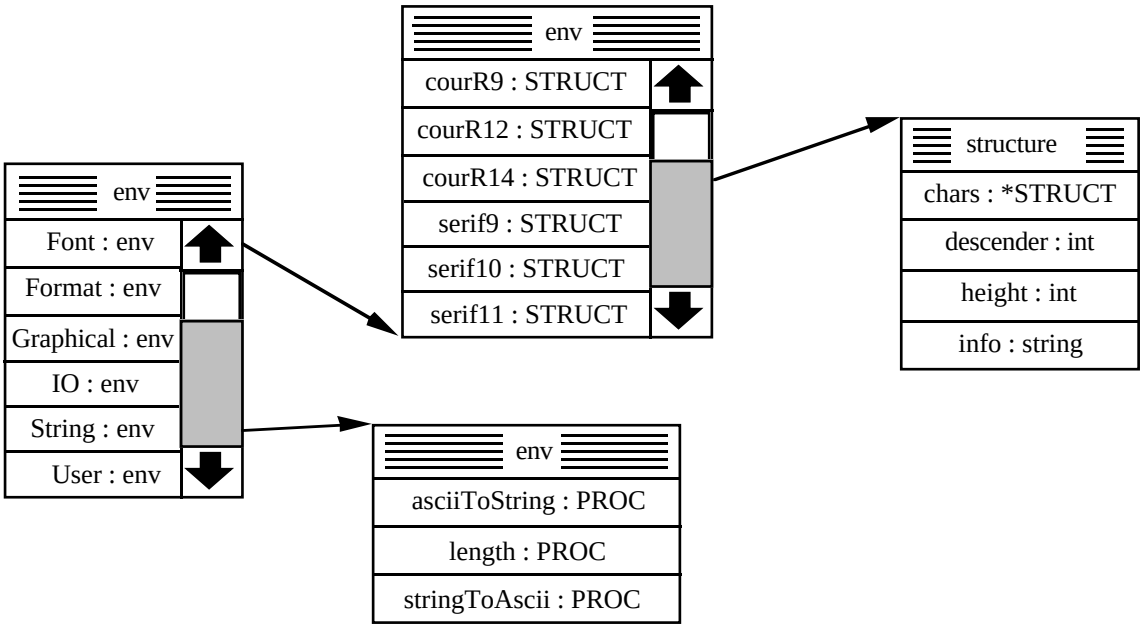

Figure 13: An Environment Hierarchy

3.2.3 Moving and Deleting Objects

When the browser is first invoked on an object, it draws the menu at the bottom left of the screen. Subsequently each object discovered by the user is drawn above and to the right of the parent object which points to it. The user can reposition objects by dragging the menu border with mouse button 2; the browser redraws any links as necessary.

The user can also remove objects from the screen by clicking mouse button 3 on the menu border. In the example of Figure 12, if the central object was deleted the arrows from the three structures would also be removed. The object could be redisplayed by selecting the ‘hd’ field of any of the structures, but the arrows to it from the other two structures would not be shown until those menu entries were selected.

3.2.4 Universes

The screen may become cluttered when the user browses a large, complex data structure. When many objects are visible simultaneously it can be difficult to see the relationships between them amid the general confusion. To aid the user in this problem, **universes** can be used to organise the data space.

To enter a new universe, the user presses mouse button 3 instead of button 1 when selecting a menu entry. The browser then creates a new window, inside which the object is displayed. In the example of Figure 11, making a new universe for the second object would result in the arrangement shown in Figure 14:

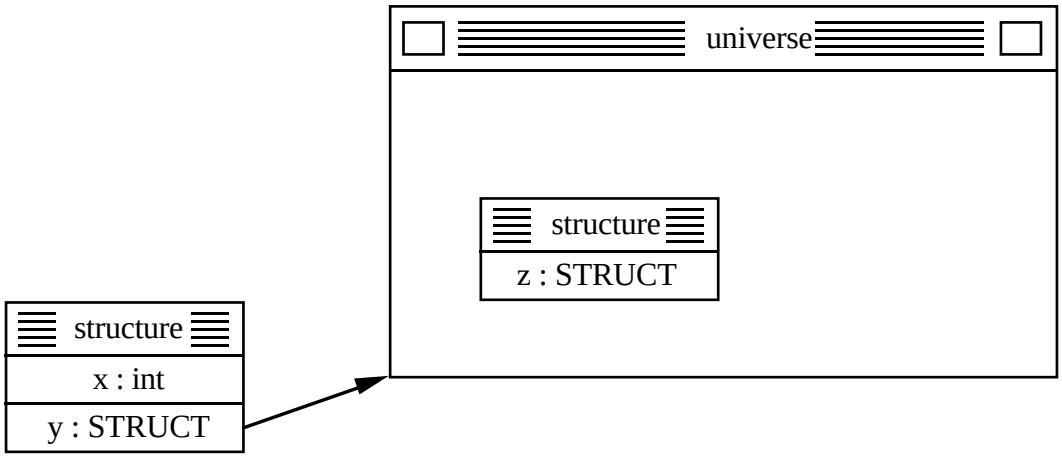

Figure 14: Component of an Object Displayed in a New Universe

The universe window can be moved around like any other object. It can also be resized by clicking mouse button 1 in the top right box and then dragging a corner of the window, and can be changed to icon form by clicking in the top left box.

The new object, the structure with field *z* in the example, is displayed by a fresh invocation of the browser which operates entirely within the universe window. Any other objects discovered from that new object will be confined to the window. In this way the object and others accessible from it are kept separate from the rest of the visible data. Universes also provide a grouping mechanism in that all the objects in a universe can be moved or deleted in one action by operating on the window containing them.

Any number of universes can be created, and they can be nested to any degree. To illustrate their usefulness, Figures 15 and 16 show a set of objects, first without any grouping, and then with partitioning into universes.

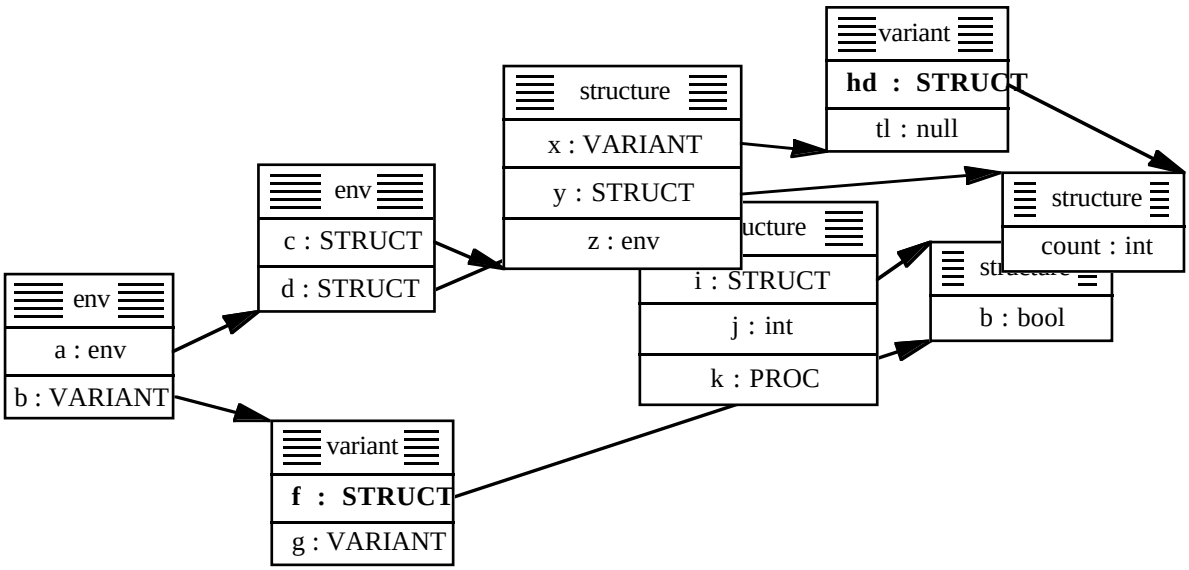

Figure 15: A Data Structure Displayed Without Universes

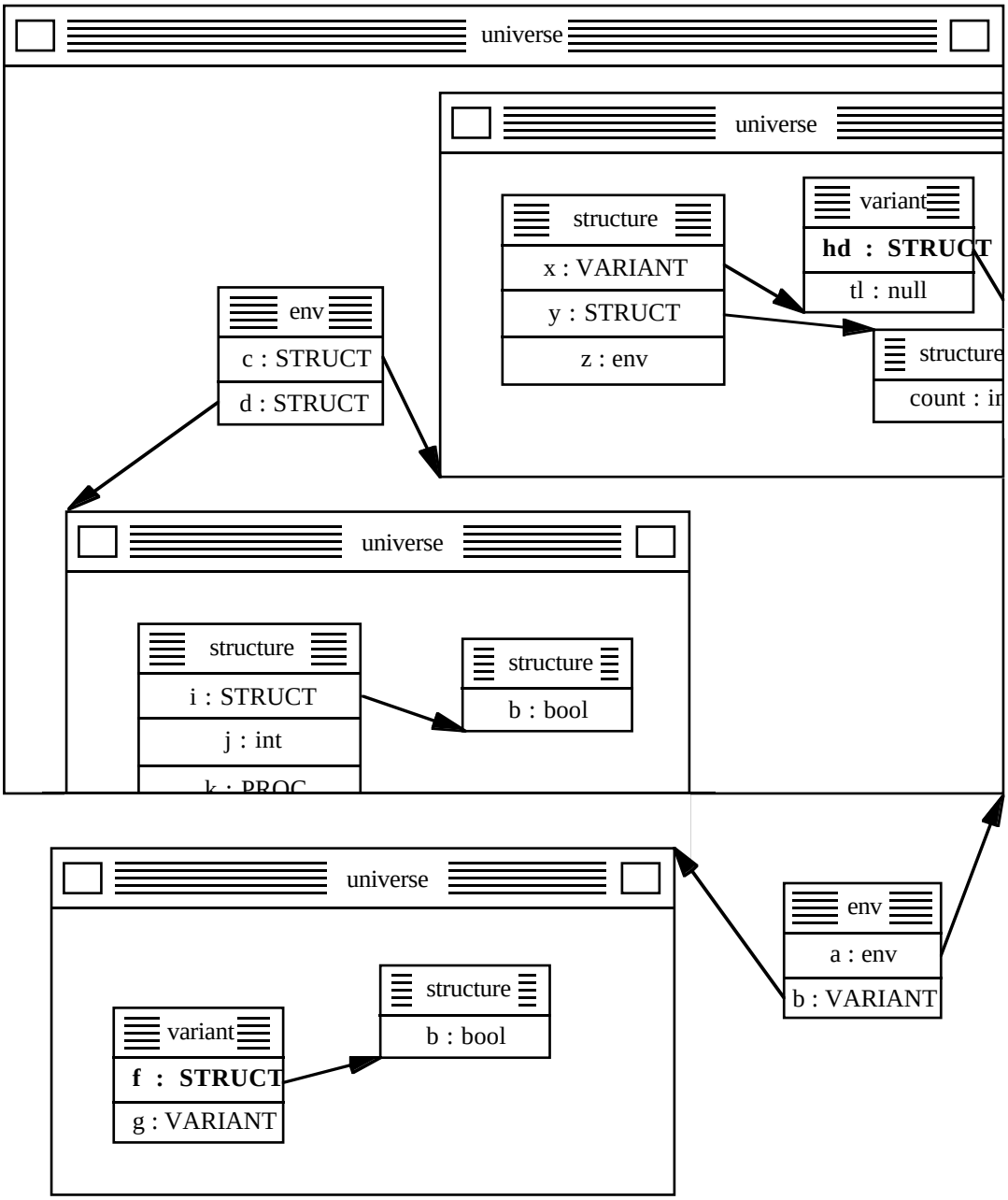

Figure 16: The Same Data Grouped into Universes

Note that some information may be lost due to the fact that universes are completely self-enclosed. In this example the browser no longer shows that the *i* field of the structure and the *f* branch of the variant both point to the same structure object. In general an object can be visible in many different universes simultaneously.

4 Implementation

This section describes the internal structure of the browser system.

4.1 How the Browser Works

The action taken by the browser depends on the type of the object encountered: it executes a different procedure for each type. As there are an infinite number of Napier88 types, it is not possible to generate all the procedures statically. The system instead relies on the Napier88 compiler being available as a procedure within the language. When the browser encounters a new type it constructs and compiles a new procedure to traverse it.

The browser maintains a table keyed by the types which have been traversed in the past, containing procedures which will traverse objects of those types. When the browser is called it first determines the type of the object inside the *any* passed to it. It then searches the table for that type, and if the type is present it calls the procedure from that table entry to traverse the object.

If, however, the type is not present in the table, the browser must construct a new procedure to traverse the object. To do this it builds a textual representation of the required procedure and then uses the callable compiler to compile the text. Before executing the procedure which has been produced, the browser enters it into the table, keyed by the type of the object which it can traverse. In this way the browser learns about new types: the next time an object of that type is encountered there will already be a procedure in the table to traverse it. Because the table is persistent, the compilation process is necessary only on the first encounter with the type.

The interface to the browser as seen by the user is a procedure of type *proc(any)*. However, internally the browser requires other information apart from the object to be traversed itself. Principally it needs to know the parent object, that is the one from which the current object was reached, in order to draw the connecting arrow correctly. To allow this, the traversal procedures which are generated and stored in the persistent table take some extra parameters.

Another constraint is that an instance of the browser has bound into it the window manager supplied when the browser was generated. However, if the traversal procedures stored in the table were also bound to a particular window manager, they would not be usable by another instance of the browser, as it might operate on a different window manager. This would mean that knowledge gained by a particular browser would persist only for that instantiation. To overcome this problem the window manager is also passed to the traversal procedures.

Each procedure is of type:

```
type Traverser is proc( any,any,env,int,int )
```

The parameters are:

- The current object to be traversed.
- The parent of the current object.
- An environment containing various data including the window manager and some user interface procedures specialised to that window manager. It also contains procedures which operate over a data structure describing the current configuration of objects and the links between them. These procedures perform such activities as drawing and deleting arrows, and finding the links to and from objects.
- The offset from the base of the parent object of the beginning of the arrow. This allows the browser to draw the arrow originating from the correct menu entry.

- The number of the field from which that arrow is drawn. This is used to distinguish any multiple links from the parent to the object. If there is already an existing link from that field then no new one will be drawn.

The browser system contains one instance of the procedure *internalTrav*, also of type *Traverser*. It is this procedure which carries out all the table look-up and compilation actions necessary for traversing an object, as described above. Each browser instance is a procedure which simply calls *internalTrav*, passing it the object to be traversed, a dummy parent object, the environment which was constructed by *travGen* when the browser was generated, and two zero values. The dummy values are used because when the browser is first invoked there is no existing parent object.

internalTrav uses the compiler implementation procedure *splitAny* to obtain a string representation of the type of the object to be traversed. Currently the compiler represents types internally by strings, so the string is obtained directly. A future version of the compiler will represent types as graphs, and then another compiler implementation procedure, *showtype*, will be used to convert the graph obtained from *splitAny* into a string.

The type string is used as the key to look up the browser's table of traversal procedures. The table is loaded at the time of its creation with procedures to traverse all base types, so if the object is of one of the base types a procedure will always be found to traverse it. If not, the string is examined to determine whether the object is a procedure, an abstract datatype, a vector, a structure or a variant. In the first two cases predefined traversal procedures are called; otherwise a new procedure is constructed, compiled, entered in the browser table, and called.

4.2 Constructing Traversal Procedures

This section gives some examples of the procedures which are constructed by the browser for traversing structures, variants and vectors.

4.2.1 Structures

Assume the following code is written by the user:

```
type X is structure( i : bool ; j : real )
type Y is structure( a : int ; b : X ; c : proc( string → int ) ; d : env )
let x = Y( 3,X( false,3.7 ),proc( s : string → int ) ; 3,PS() )
trav( any( x ) )
```

Figure 17 shows the procedure which is constructed by the browser:

```
rec type t0 is structure( a : int ; b : t1 ; c : t2 ; d : env )
&        t1 is structure( i : bool ; j : real )
&        t2 is proc( string → int )

use PS() with IO,Lib : env in
use Lib with browser2 : env in
use browser2 with internalTrav : Traverser in

proc( object,parent : any ; local : env ; menuOffset,fieldNo : int )

use local with                    error :      proc( string );
               addReference :     proc( any,any,int );
               connect :          proc( any,any,bool,int );
               objectReferenced : proc( any,any → bool );
               objectDisplayed :  proc( any → bool );
               showMenu :         proc( any,any,string,*string,*bool,*proc( int ) );
               newUniverse :      proc( any,any,int ) in
begin
    let doMenu = proc( value : t0 ; parent : any )
        showMenu( any( value ),
                  parent,
                  "structure",
                  vector @1 of [ "a : int","b : STRUCT","c : PROC","d : env" ],
                  vector @1 of [ false,false,false,false ],
                  vector @1 of [      proc( button : int )
                                      if button = 1
                                      then internalTrav( any( value( a ) ),any( value ),local,8,1 )
                                      else newUniverse( any( value( a ) ),any( value ),8 ),

                                      proc( button : int )
                                      if button = 1
                                      then internalTrav( any( value( b ) ),any( value ),local,24,2 )
                                      else newUniverse( any( value( b ) ),any( value ),24 ),

                                      proc( button : int )
                                      if button = 1
                                      then internalTrav( any( value( c ) ),any( value ),local,40,3 )
                                      else newUniverse( any( value( c ) ),any( value ),40 ),

                                      proc( button : int )
                                      if button = 1
                                      then internalTrav( any( value( d ) ),any( value ),local,56,4 )
                                      else newUniverse( any( value( d ) ),any( value ),56 ) ] )

    project object as X onto
    t0 :  if ~objectReferenced( object,parent ) do          !*** Already a link from the parent to this object?
          begin
              if ~objectDisplayed( object ) do              !*** Object already displayed but no link from parent?
                  doMenu( X,parent )                        !*** Object not displayed yet.
              addReference( parent,object,menuOffset )
              connect( parent,object,true,menuOffset )
          end
    default : error( "wrong traverse procedure called." )
end
?
```

Figure 17: Browser Code Generated for a Structure

4.2.2 Variants

Assume the following code is written by the user:

```
type W is variant( i : int ; j : real )
type X is abstype[ I ]( x : I )
type Y is variant( a : W ; b : X ; c : image )
let x = Y( a : W( j : 3.14 ) )
trav( any( x ) )
```

Figure 18 shows the procedure which is constructed by the browser:

```
rec type t0 is variant( a : t1 ; b : t2 ; c : image )
&        t1 is variant( i : int ; j : real )
&        t2 abstype[ t3 ]( x : t3 )

use PS() with ... in
begin
    let getBranch = proc( X : t0 → int )
        project X as Y onto
        a : 1
        b : 2
        c : 3
        default : 0

    proc( object,parent : any ; local : env ; menuOffset,fieldNo : int )
    use local with ... in
    begin
        let doMenu = proc( value : t0 ; parent : any )
        begin
            let nothingProc = proc( i : int ) ; {}                    !*** Define proc which does nothing.
            let branchNo = getBranch( value )                         !*** Find which branch is present.
            showMenu( any( value ),parent,"variant",
                      vector @1 of [ "a : VARIANT","b : ABSTYPE","c : image" ],
                      vector @1 of [ branchNo = 1,branchNo = 2,branchNo = 3 ],
                      vector @1 of [      if branchNo = 1
                                          then    proc( button : int )
                                                  if button = 1
                                                  then internalTrav( any( value'a ),any( value ),local,8,0 )
                                                  else newUniverse( any( value'a ),any( value ),8 )
                                          else nothingProc,

                                          if branchNo = 2
                                          then    proc( button : int )
                                                  if button = 1
                                                  then internalTrav( any( value'b ),any( value ),local,24,0 )
                                                  else newUniverse( any( value'b ),any( value ),24 )
                                          else nothingProc,

                                          if branchNo = 3
                                          then    proc( button : int )
                                                  if button = 1
                                                  then internalTrav( any( value'c ),any( value ),local,40,0 )
                                                  else newUniverse( any( value'c ),any( value ),40 )
                                          else nothingProc ] )
        end

        project object as X onto
        t0 :  if ~objectReferenced( object,parent ) do        !*** Already a link from the parent to this object?
              begin
                  if ~objectDisplayed( object ) do            !*** Object already displayed but no link from parent?
                      doMenu( X,parent )                      !*** Object not displayed yet.
                  addReference( parent,object,menuOffset )
                  connect( parent,object,true,menuOffset )
              end
        default : error( "wrong traverse procedure called." )
    end
end
?
```

Figure 18: Browser Code Generated for a Variant

4.2.3 Vectors

Assume the following code is written by the user:

```
type X is structure( a : int )
let x = vector 1 to 10 using proc( i : int → X ) ; X( i )
trav( any( x ) )
```

Figure 19 shows the procedure which is constructed by the browser:

```
type t0 is  structure( a : int )                    !*** Define the type the proc works for.

use PS() with … ;
            checkNumber :  proc( string → bool );   !*** Returns true if the string has only digits.
            stringToInt :  proc( string → int ) in  !*** Converts string containing digits to an int.

proc( object,parent : any ; local : env ; menuOffset,fieldNo : int )

use local with … in
begin
    let doMenu = proc( value : *t0 ; parent : any )
    begin
        let lowerBound = lwb[ t0 ]( value )
        let upperBound = upb[ t0 ]( value )
        let title = "vector"
        let entries = vector @1 of [ "lwb","upb","index","list" ]
        let bold =  vector @1 of [ false,false,false,false ]
        let actions =           vector @1 of [   proc( i : int ) ; outputString( "lwb: " ++ iformat(
lowerBound ) ++ "'n" ),
                                proc( i : int ) ; outputString( "upb: " ++ iformat( upperBound ) ++ "'n" ),
                                proc( button : int )
                                begin
                                    let travIndex = proc( s : string )
                                    if checkNumber( s ) do
                                    begin
                                        let index = stringToInt( s )
                                        if index >= lowerBound and index <= upperBound do
                                            if button = 1
                                            then internalTrav( any( value( index ) ),any( value ),local,-1,0
)
                                            else newUniverse( any( value( index ) ),any( value ),-1 )
                                    end
                                    dialogue( "index: ",travIndex )
                                end,
                                proc( button : int )
                                    for i = lowerBound to upperBound do
                                        if button = 1
                                        then internalTrav( any( value( i ) ),any( value ),local,-1,0 )
                                        else newUniverse( any( value( i ) ),any( value ),-1 ) ]
        showMenu( any( value ),parent,title,entries,bold,actions )
    end

    project object as X onto
    *t0 :  if ~objectReferenced( object,parent ) do     !*** Already a link from the parent to this object?
           begin
               if ~objectDisplayed( object ) do         !*** Object already displayed but no link from parent?
                   doMenu( X,parent )                   !*** Object not displayed yet.
               addReference( parent,object,menuOffset )
               connect( parent,object,true,menuOffset )
           end
    default : error( "wrong traverse procedure called." )
end
?
```

Figure 19: Browser Code Generated for a Vector

4.3 Code for the Main Browser Procedure

Figure 20 illustrates the way in which the browser uses the procedures it creates to traverse objects, with the code for the procedure *internalTrav*:

```
!*** Type of traversal procedures stored by browser.
type Traverser is proc( any,any,env,int,int )

!*** Generic type of table holding traversal procedures.
!*** Browser uses instance specialised to types [ string,Traverser ]
type Result[ Data ] is variant( ok : Data ; fail : null )
type Table[ Key,Data ] is structure(        enter :    proc( Key,Data );
                                    lookup :   proc( Key → Result[ Data ] );
                                    remove :   proc( Key );
                                    scan :     proc( proc( Key,Data → bool ) );
                                    firstKey : proc( → Result[ Key ] ) )

!*** This procedure uses the procedures splitAny and compile.
!*** splitAny :  proc( any → structure( pointer : null ; tr : typeRep ; branch,padding : int ) )
!***            type typeRep is structure( unboundQuantifiers : bool ; representation : string )
!*** compile :  proc( string → any )

let internalTrav = proc( object,parent : any ; local : env ; menuOffset,fieldNo : int )

use browser2 with travProcsTable : Table[ string,Traverser ] in
use local with                   windowManager : WindowManager;
               error :           proc( string ) in
begin
    let split = splitAny( object )                  !*** Break the any into its components.
    let stringRep = split( tr )( representation )   !*** String representation of the type.

    let try = travProcsTable( lookup )( stringRep ) !*** See if it is in the table.

    if try is ok then
    begin
        !*** A procedure to traverse this type already exists.

        let traverseObject = try'ok
        traverseObject( object,parent,local,menuOffset,fieldNo )    !*** Use the traversal procedure found.
    end
    else
    begin
        !*** Haven't encountered this type before.  See what sort of object it is.
        !*** It can't be a base type as an entry in the table would then have been found.

        let kind = kindOf( stringRep )    !*** Returns one of "proc", "abstype", "struct", "variant" or "vector".

        case kind of
        "proc":      traverseProc( object,parent,local,menuOffset,fieldNo )
        "abstype":   traverseAbstype( object,parent,local,menuOffset,fieldNo )

        "struct":
        begin
            !*** Construct a new procedure to traverse this kind of structure.
            let travProcText = traverseStruct( stringRep )    !*** Make text representation of new proc.
            let travProc = compile( travProcText )            !*** Compile it.

            project travProc as trav onto                     !*** Check proc is of correct type.
            Traverser :
            begin
                travProcsTable( enter )( stringRep,trav )     !*** Enter it in table for future use.
                trav( object,parent,local,menuOffset,fieldNo ) !*** Use it.
            end
            default : error( "compilation of structure traverser failed." )
        end
```

```
            "variant" :
            begin
                  !*** Construct a new procedure to traverse this kind of variant.
                  let travProcText = traverseVariant( stringRep )
                  … Code as for structure …
                  default : error( "compilation of variant traverser failed." )
            end

            "vector" :
            begin
                  !*** Construct a new procedure to traverse this kind of vector.
                  let travProcText = traverseVector( stringRep )
                  … Code as for structure …
                  default : error( "compilation of vector traverser failed." )
            end

            default :   error( "object of unknown sort." )
      end
end
```

Figure 20: Code for the Procedure *internalTrav*

5 A Faster Implementation

5.1 Performance of the Original Implementation

The browser implementation we have described works correctly, and is interesting as an exercise in reflective programming: the system manipulates text to construct new Napier88 procedures, using the language itself. Its main deficiency, however, is its performance. Whenever an unfamiliar type is encountered some compilation must be carried out, often resulting in an uncomfortable delay for the user. For structures, variants and vectors, at least, the compilation phase is only necessary on the first encounter with the type. However, to traverse an environment the browser must compile one procedure to access each of the bindings within it. This is a consequence of the fact that the bindings are from values to names rather than to strings. To write code to look up a binding, the programmer must know its name at compile time, whereas the browser may be presented with environments containing arbitrary names which cannot be discovered until run time.

The solution used is to scan the environment at run time, and then to bind the names discovered into textual representations of procedures which perform the look-ups. The first time a particular binding is accessed, the code for the appropriate procedure is compiled and then stored for reuse if the binding should be accessed again. Unlike the procedures which are compiled for traversing structures, variants and vectors, the look-up procedures persist only as long as the environment menu is displayed. If the menu is removed and the environment then traversed again, the procedures will be constructed and compiled again.

Environment traversal is thus especially prone to performance problems. If the browser system is to be a useful tool, its speed of operation must somehow be improved. This is possible so long as we are willing to sacrifice the reflective nature of the system, and to build it at a lower level which involves knowledge of the formats of objects at the store level. The method used in the most recent implementation of the browser relies on the manipulation of the concrete representations of language objects. It gives much improved response times, with the disadvantage, however, that the browser is now tied to a particular implementation of the Napier88 system.

5.2 Type Magic Procedures

This section describes the procedures which perform type magic and are used in the browser implementation described in the following sections. These procedures are available only in the boot version of the Napier88 compiler.

coerceFromEnv : **proc**(**env** → EnvImplementation)

This returns a structure containing the procedures implementing the given environment.

makeObject : **proc**(**int**,**int** → **null**)

This returns a pointer to a new store object with the given size and number of pointer fields.

formAny : **proc**(**null**,typeRep,**int**,**int** → **any**)

This builds an instance of type *any* from its components. The components consist of a pointer to the object to be injected, wrapped in a store object if necessary, a representation of the type of the object, the branch number, and a padding word which is ignored. The system representation of *any*s is described in greater detail in the PAM Manual [BCC88].

splitAny : **proc**(**any** → **structure**(pointer : **null** ; tr : typeRep ; branch,padding : **int**))

This splits an any into its components as described above.

assignPntr : **proc**(**null**,**int**,**null**)

This assigns a pointer to the specified position in the given store object.

assignInt : **proc**(**null**,**int**,**int**)

This assigns a non-pointer to the specified position in the given store object.

lookupPntr : **proc**(**null**,**int** → **null**)

This reads a pointer from the specified position in the given store object.

lookupInt : **proc**(**null**,**int** → **int**)

This reads a non-pointer from the specified position in the given store object.

5.3 Faster Traversal Methods

This section describes the methods used to traverse environments, structures, variants, vectors and procedures.

5.3.1 Environments

An environment is represented in the current Napier88 system by a structure containing procedures for adding, looking up and removing bindings, scanning the environment, etc.. The procedures have encapsulated in their closures some data structure, such as a binary tree or a hash table, within which the environment's bindings are stored. Strings containing the binding names are used to index the bindings. The environment representation has the following type:

```
type envImplementation is structure(    add : proc( string,typeRep,bool,null );
                                        delete : proc( string );
                                        find : proc( string,typeRep,bool → bool );
                                        lookup : proc( string,typeRep,bool → null );
                                        scan : proc( proc( string,typeRep,bool ) ) )
```

typeRep is the type used to represent types by the Napier88 run-time system. Within the system, pointers to objects in the Napier88 store are represented as values of type *null*. These values are not visible to the user, to whom it appears that *nil* is the sole instance of type *null*. The *null* parameters to the procedures above correspond to pointers to the objects stored in the bindings. For each binding of a type whose instances are represented by a single pointer, such as a structure, a vector or an image, the environment implementation points directly to the object in the binding. For all other types it points to a store object in which the value is wrapped. The boolean parameters to the procedures correspond to the constancies of the bindings.

To traverse an environment, the browser first obtains its implementation structure, using the procedure *coerceFromEnv* (described in *Section 5.3 Magic Procedures*). It uses the *scan* procedure in the structure to get the names of all the bindings present, as strings, and the corresponding type representations. These are used to construct the entries for the menu.

When a particular entry is selected by the user, the browser must extract the required value from the environment implementation, and convert it to type *any* so that it can be passed back to the browser. A pointer to the value (or the object in which it is wrapped) is obtained using the *lookup* procedure. The browser then uses the procedure *formAny* to make it into an instance of type *any*. This procedure takes as its parameters the pointer to the object or wrapper, its type (which was discovered when the original scan of the environment took place), and an integer tag. This tag, the *machine tag*, relates to the type system used within the Napier88 Persistent Abstract Machine [BCC88],[CBC89] and its value encodes the number of pointer and non-pointer words required to represent the object. The value is fixed for each base type and type constructor, so that there is, for example, one tag value for reals and another for structures.

In summary, when the browser traverses an environment it initially:

- calls *coerceFromEnv* to obtain the procedures which implement the environment;
- calls one of those procedures to scan the environment to discover the name and type of all the bindings.

To extract the value from a particular binding it:

- calls another implementation procedure to obtain a pointer to the object or its wrapper;
- calculates the machine tag from the type of the object;
- calls *formAny* to turn the pointer, type and tag into an *any*;
- calls itself to traverse the *any*.

The initial phase is essentially the same as that carried out in the previous browser implementation. The process of accessing a binding in the environment, however, is much faster than the compilation which was previously necessary.

5.3.2 Structures

To traverse a structure the browser must be able to build a menu showing the field names and their types, and extract the value stored in any field of the structure. In the previous implementation it was necessary to compile code to dereference each field of the structure, for the same reason that compilation was necessary for environments: the name of the field to be dereferenced must be known statically. Again, the alternative to writing code to perform the dereference is to obtain the representation of the structure and to look up the value directly.

For structures, the names of the fields and their types are obtained from the representation of the type, as opposed to a scan of the object for environments. That scan must take place every time an environment is traversed, whereas the analysis of a structure type is necessary only on the first encounter with the type. On that first encounter, the browser uses *splitAny* to obtain a pointer to the structure object and a representation of its type. It then uses the type representation to construct further type representations, one for each of the structure fields.

As well as knowing the type of a field, the browser must also be able to extract its value, by reading words from the appropriate positions within the structure object. All structures have the same general store format in the current Napier88 implementation: pointers come before non-pointers, and subject to that constraint the fields are arranged in alphabetic order. For fields of types which have both pointer and non-pointer components, the pointer words may be in a different part of the object from the non-pointers. However, the positions of the components for each field are constant for a given structure type, so it is only necessary to calculate them once, at the same time that the field type representations are calculated.

To calculate the offsets within the object for each field, the browser calculates the number of pointer and non-pointer words taken up by each of the fields. These, and the machine tag which is also required to turn the field value into an *any* (as in environment traversal described earlier), depend on the type of the field. All three values are calculated by a simple case switch on the type name if it is a base type, or the type constructor otherwise. Once the sizes of all the fields have been determined, and using knowledge of the general structure format, the browser builds a map which describes the positions of the pointer and non-pointer components of each of the fields. This map can then be used to find the value of any field of any instance of the structure type.

The browser stores a procedure in the table of traversal procedures, with the map, the field type representations and the machine tags bound into its closure. When the procedure is called to traverse an instance of the structure type, it creates a menu with an entry for each field. Each of the procedures associated with a particular menu entry will have bound into it the location of the value of that field within the structure, obtained from the map, as well as the type of the field, and its machine tag. When one of those procedures is itself called, to traverse a field of the structure, it looks up the value of the field and converts it, along with the type and the tag, into an *any*. The procedure used for this, *formAny*, must be passed a pointer to a store object, so if the value is not itself a pointer, a new store object must be created, the value copied into it, and a pointer to the wrapper object used.

The actions performed by the browser are thus as follows. On encountering an object with an unknown structure type it:

- obtains a representation of the structure type by passing the object to *splitAny*;
- uses that type representation to construct representations of the types of all the fields;
- uses the types of the fields to work out the machine tag and the size of the pointer and non-pointer components for each field.

To traverse one of the fields of a structure of that type it:

- obtains a pointer to the store representation of the object by passing the object to *splitAny*;
- looks up the offsets of the pointer and non-pointer components of the required field from the map constructed previously;
- reads the values of the components from the appropriate positions within the object using *lookupInt* and *lookupPntr*;
- encloses the components in a wrapper object if necessary, using *makeObject*, *assignInt* and *assignPntr*;
- converts the field value to an *any* by calling *formAny* with the value, the type and the machine tag for the field;
- calls itself to traverse the *any*.

The pay-off in improvement in performance comes at a different point from that in environment traversal. In the original implementation the display of an environment menu was cheap, but it was expensive to traverse a value in the environment. It would have been possible to shift much of that cost to the point where the menu was displayed, but this would probably not be sensible as the user might well traverse only a few of the bindings. There is little change in the cost of the initial display in the new implementation, but it has become much faster to examine a binding.

However, traversing the fields of a structure of known type was already reasonably fast, and this has not changed. The improvement for structures is a reduction in the work necessary when the browser first encounters an unfamiliar type. Unlike the case for environments, it is desirable for the browser to perform more work on the first encounter with a structure type if this will reduce the cost of accessing the field values. This is because the benefit extends to other instances of the type, whereas the names and types of the bindings in an environment must be discovered anew each time it is presented to the browser.

5.3.3 Variants

Variants are handled in a similar way to structures. When an unfamiliar variant type is encountered, the browser determines the type representations and machine tags for each of the possible branches. As an instance of a variant contains only one branch, there is no problem locating the injected value within the variant.

Each time a variant of that type is traversed, the browser must decide which branch of the variant is present, in order to display the correct menu entry in bold. This information is encoded within the branch number obtained when the variant is passed to *splitAny*, and can be recovered with a simple bit-wise shift.

In summary, the actions in traversing a variant are as follows. On encountering an object with an unknown variant type the browser:

- obtains a representation of the variant type by passing the object to *splitAny*;
- uses that type representation to construct representations of the types of all the branches;
- uses the types of the fields to work out the machine tag for each branch.

To traverse one of the branches of a variant of that type it:

- passes the object to *splitAny* to obtain a pointer to the injected value, and the *any* branch number;
- calculates which variant branch is present from the *any* branch number;
- if the branch requested is the one present, converts the injected value to an *any* by calling *formAny* with the value, the type and the machine tag for the field;
- calls itself to traverse the *any*.

As with structures, the speed-up with this implementation is obtained at the point of the first encounter with a variant type, rather than at the traversal of the branches of other instances of the type.

5.3.4 Vectors

Vector traversal is also similar to structure traversal. It is simpler in that all the elements of a vector have the same type, so the browser has only one type representation to construct when it first encounters a vector type. At this stage the browser also determines the machine tag of the elements, and the size of their pointer and non-pointer components.

As for structures and variants, the browser enters in the traversal table a procedure to construct menus for other instances of the vector type. To do this, the procedure must generate further procedures to display the lower and upper bounds of the vector, and to traverse elements of the vector. The bounds are determined by looking up the last two words of the vector object, where the vector bounds are stored in the current Napier88 implementation. To traverse an element, the browser calculates the positions of its components within the vector object, from the size of the vector elements and the index of the element. It then reads the components and converts them into an *any* in the same way it does for a structure field.

A summary of the steps in vector traversal follows. On encountering a vector with an unknown type the browser:

- obtains a representation of the vector type by passing the vector to *splitAny*;
- constructs from it the type representation of the vector elements, and calculates the machine tag value and the size of the pointer and non-pointer components.

When traversing a vector of that type it:

- obtains a pointer to the vector object by passing the vector to *splitAny*;
- determines the vector bounds by reading from the last two words in the vector object, using *lookupInt*; or
- calculates the positions within the vector object of the components of a particular element;
- reads the values of the components from the object using *lookupInt* and *lookupPntr*;
- encloses the components in a wrapper object if necessary, using *makeObject*, *assignInt* and *assignPntr*;
- converts the element value to an *any* by calling *formAny* with the value and the element type and machine tag;
- calls itself to traverse the *any*.

Again, the gain in speed comes when the browser first encounters a vector of a particular type.

5.3.5 Procedures

Procedure traversal is available only in the latest implementation of the browser, as it relies heavily on the magic procedures described in the following section. The browser first determines whether the procedure passed to it has PAIL[1] attached. It does this by calling *splitAny* to obtain the wrapper object for the procedure closure, and then using *lookupPntr* to read pointers to the code vector and the static link from the wrapper. It then reads the PAIL slot in the code vector: if the value there is *nil* it just writes a message to the text output window.

If the PAIL is present, indicated by a non-*nil* pointer in the code vector, the browser converts the pointer of type *null* into an object of the variant type PAIL. The pointer points to a wrapper for the components of the variant; the browser reads these using *lookupInt* and *lookupPntr*, and then passes them, together with a pre-constructed type representation for PAIL, to *formAny*. Projecting from the *any* produced results in a PAIL object.

The browser then performs several scans over the PAIL, the first of these to find any type definitions. It recursively traverses the tree, examining all *Link* structures. A *Link* is a part of the PAIL corresponding to a declaration of an object or a type. Whenever the browser finds a *Link* whose *StoredType* field holds the value *true*, indicating a type declaration, it looks up the name of the type and its definition, and enters them in a table keyed by the type. Types are represented by graphs within PAIL, so the browser has to define an ordering over those graphs so as to be able to use types as table keys. This was done previously in the Napier88 type checker [Con88].

The browser also scans to find the *Link*s for all the objects referred to in the procedure code. This time it adds to a list all the *Link*s whose *Addr* field takes the *Stack* branch, indicating that the *Link* represents a stack address.

The browser shows a reconstruction of the procedure code in the main area of the procedure window. This is generated by another recursive scan of the PAIL. When the browser has to show a type, such as in a procedure heading, it attempts to look up the type in the table previously constructed. If the type is present, it uses the corresponding name; if not, it gives the type a name such as *t0, t1* etc., and generates a string representation of the type using the compiler procedure *showtype*. It then enters a definition of the type in the lower area of the window.

Finally, the browser constructs an environment containing the values of all the non-local objects which are referred to in the procedure code but not defined there. To do this it reads the stack address in each *Link*, which gives a position within the static chain and the offsets within the frame for both pointer and non-pointer components. For those addresses which are not in the local frame, the browser follows the static chain to find the correct frame, looks up the components, wraps them if necessary, and places them in the environment using the environment's implementation procedures. If the number of entries is non-zero, it then displays a menu for the environment in the procedure window.

6 Future Directions

There are several areas in which improvements and developments could be made:

- Menus could be positioned more intelligently when the browser first places them on the screen. Currently each new menu is initially displayed at a fixed offset to its parent menu, even when that results in it disappearing off the edge of the screen.

---

[1]PAIL is described in [Dea87] but note that this covers the original version as used for PS-algol, and not the Napier88 version.

- Displaying of abstypes. Currently the system just displays the message ‘an abstype’, but it would be better to show the type of the object: this would require some mechanism for browsing types rather than values. This could be done by showing menus similar to those used currently. Selecting an entry with a scalar type would have no effect, while selecting a complex type would bring up another menu showing the types of its elements.

- The table of traversal procedures is currently keyed by the string representation of the type traversed, and these strings are potentially very long, so table lookup may become expensive. If a compiler with graph representations of types was used, the table could be keyed using the graphs. This would require an arbitrary ordering on the graphs to be defined (a problem which has already been addressed in procedure traversal). Such an ordering would make comparison of highly dissimilar types very fast – but then this is also the case for string comparison. This matter requires experimentation.

- Facilities for changing values, rather than just examining them, could be provided. These could include the abilities to drop bindings from environments, introduce new bindings into environments, and assign new values to variable locations within structures and environments. At a simple level the user could write a textual description of any new value, which would then be compiled. More desirable would be the ability to construct new values from old using a combination of mouse gesture and text input. Given that it is vital to maintain the integrity of the persistent store, we consider that the user should not be allowed to change anything using the browser that he or she could not do using a Napier88 program.

7 References

[BCC88] Brown A.L., Connor R.C.H., Carrick R., Dearle A. & Morrison R.
"The Persistent Abstract Machine",
PPRR-59, Universities of Glasgow and St Andrews, Scotland (1988).

[CBC89] Connor R.C.H., Brown A.L., Carrick R., Dearle A. & Morrison R.
"The Persistent Abstract Machine",
Proc. Third International Conference on Persistent Object Systems,
Newcastle, Australia, pp. 80-95 (1989).

[CDK89] Cutts Q.I., Dearle A., Kirby G.N.C. & Marlin C.D.
"WIN: A Persistent Window Management System",
PPRR-73, Universities of Glasgow and St Andrews, Scotland (1989).

[CDK90] Cutts Q.I., Dearle A. & Kirby G.N.C.
"WIN Programmer's Manual",
Available from the authors of this paper.

[Con88] Connor R.C.H.
"The Napier Type-Checking Module",
PPRR-58, Universities of Glasgow and St Andrews, Scotland (1988).

[DB88] Dearle A. & Brown A.L.
"Safe Browsing in a Strongly Typed Persistent Environment",
Computer Journal Vol 31, No 6 (1988).

[DCK89] Dearle A., Cutts Q.I. & Kirby G.N.C.
"Browsing, Grazing and Nibbling Persistent Data Structures",
Proc. Third International Conference on Persistent Object Systems,
Newcastle, Australia, pp. 96-112 (1989).

[Dea87] Dearle A.
"A Persistent Architecture Intermediate Language",
PPRR-35, Universities of Glasgow and St Andrews, Scotland (1987).

[Dea88] Dearle A.
"On the Construction of Persistent Programming Environments" (PhD Thesis),
PPRR-65, Universities of Glasgow and St Andrews, Scotland (1988).

[GR83] Goldberg A. & Robson D.
"Smalltalk-80: the Language and its Implementation",
Addison-Wesley (1983).

[KD90] Kirby G.N.C. & Dearle A.
"An Adaptive Browser for Napier88",
Available from the authors of this paper.

[MBC89] Morrison R., Brown A.L., Connor R.C.H. & Dearle A.
"The Napier88 Reference Manual",
PPRR-77, Universities of Glasgow and St Andrews, Scotland (1989).

[PS87] "The PS-algol Reference Manual, Fourth Edition",
PPRR-12, Universities of Glasgow and St Andrews, Scotland (1987).