This paper should be referenced as:

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S. & Morrison, R. “Unifying Interaction with Persistent Data and Program”. In **Interfaces to Database Systems**, Sawyer, P. (ed), Springer-Verlag, Proc. 2nd International Workshop on User Interfaces to Databases, Ambleside, Cumbria, 1994 (1995) pp 197-212.

Unifying Interaction with Persistent Data and Program

R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby,
V.S. Moore and R. Morrison

Division of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife KY16 9SS, Scotland

Abstract

Visual interaction with object-oriented databases, such as that provided by generic object browsing systems, has proved to be a convenient and natural way for database users to address informal queries over the contents of a database. Our particular field of interest is browsing and editing in persistent and database programming languages where procedures are treated as data values, with the consequence that executable code may exist in the same persistent environment as the other data that it manipulates. Such systems include object-oriented database systems, where the objects' method code is an intrinsic part of the object database itself.

A new style of browsing is introduced which allows a browser/editor to subsume all the activities normally connected with writing queries and other programs against the database. It therefore provides the only interface to the database that programmers and users require to understand. This is achieved partly by unifying the concepts of source and executable code within a system. This unification relies upon the paradigm of hyper-programming, in which programs may contain direct links to database values embedded in their source representations.

1 Overview

This paper is about a new style of browser/editor which allows programmers and users to interact directly with the database, by allowing database accesses to be performed by user gesture. Browsing a database in this style is a well-established technique; the contribution here is an extension of this browsing style which allows the programs which operate over the data to be traversed and manipulated in the same manner as other database values.

Our field of interest is in persistent and database programming languages where procedures are treated as data values, with the consequence that executable code may exist in the same persistent environment as the other data that it manipulates. Such systems include object-oriented database systems, where the objects' method code is an intrinsic part of the object database itself.

The major conceptual difference between procedures and methods, that of late binding to object instances, is unimportant in the context of this discussion. The binding issues which will be discussed, notably those of closure formation, are identical in the different paradigmatic contexts of object-oriented databases and other persistent and database programming languages. The examples in the paper will be given in the context of a procedural database programming language (in fact Napier88 [1]), but the concepts extend to any database system in which the code is conceptually resident in the object store.

2 Introduction

Visual interaction with object-oriented databases, such as that provided by generic object browsing systems [2-8], has proved to be a convenient and natural way for database users to address informal queries over the contents of a database. The users of such tools can browse freely around the data structures and values of a database, avoiding the necessity to write down algebraic expressions to perform the equivalent accesses. Where appropriate it is also possible to perform updates or invoke more complex methods over the objects depicted on the screen. Such tools are greatly preferred to a traditional query-based approach for simple queries and updates to object-oriented databases.

The advantages of this style of access are comparable to the advantages of a modern iconic operating system interface over a traditional command-line based approach. In addition, however, a more general programming algebra is required so that more complex and longer-running queries may be handled. This rather frustratingly gives rise to two quite separate mechanisms for manipulating the same values within a system, with the choice of mechanism being somewhat arbitrary for tasks in the middle ground between trivial and complex.

Here we describe a new browser/editor that is being developed for the Napier88 system [9]. The significance of this new browser is that it includes three very strong unifying concepts, the combination of which makes the browser the only mechanism which is required for interaction with the database system. The three important unifying concepts are:

1. Data of any type supported by the system may be browsed and edited in a uniform manner. This includes a uniform treatment of procedure closures; a drawback of previous browsers is that they could not adequately handle procedures.
2. Source code is treated not as a fundamental building block within the programming system, but instead as a transient text-based view of any value. The source does not have a conceptual permanent existence within the system, but is apparently generated from any value that may be browsed. Compilation and linking still occur within the system, but are presented to the programmer simply as a reification of the transient source.
3. As a further consequence of the generic treatment of procedure values and source code, the artificial distinction between source and executable values

within a running system is completely removed. If a run-time error occurs, for example, the actual value of the procedure in which it occurred may be displayed by the browser. The source will still be available for necessary purposes such as correction and adaptation; however the source will not exist as a separate entity accessible in isolation from the executable value.

The major difference between this and other browsers is therefore in the uniform treatment of the executable and source code forms of procedures, and hence programs. Furthermore, as will be seen, the manipulation of code made possible by the unification strategy is sufficiently general to subsume the usual process of program editing, compilation and linking which is normally associated with the manipulation of code bodies within a system. This means that the browser/editor is the only interfacing tool required to perform queries of any complexity against the database, or to introduce new data and program to it.

The unification of the source and executable code within a system is not possible with a conventional source representation. Section 3 explains the basic problems, and highlights a new style of source representation, that of hyper-programming, as a way of solving them. Section 4 then introduces the main concepts of the new-look browser/editor.

3 Unifying Source and Executable Forms

This section examines the essential problem with the uniform visualisation of persistent source and executable code representations; namely, that many different executable forms may share the same source representation. It is shown how the paradigm of hyper-programming, where direct links to values may exist as part of the source code, may be used to create a one-to-one mapping between source and executable, with the direct consequence that the concepts may be merged in the programmers' view of the system.

3.1 Code and Closure Representations

There is normally a clear distinction in persistent systems which support first-class procedures between the concept of procedure code, and the concept of closure; it is this distinction which traditionally makes the unification of source and executable values impossible. The concept of closure represents the executable version of code; it is formed by a pair consisting of the procedure code and the environment in which this code is to be evaluated. Thus the difference between code and closure can be summed up as the meaning of any free variables in the code at the time the procedure value is instantiated.

For a procedure without free variables, the environment is not significant and there is no conceptual difference between code and closure except for the fact that one form is regarded as source and the other as executable. The system's compiler performs a mapping from one form to the other; it is quite possible to construct a reverse mapping from executable to source. This is true in any system, and indeed many systems do keep such reverse mappings for debugging purposes. If all proce-

dures within a system have the property of containing no free variables, then this mapping is one-to-one; that is, no two different executable forms map to the same source form. This is a direct consequence of the executable semantics being wholly captured by the procedure code. If all the code in the system may be defined in terms of such source-executable pairs, then the conceptual unification of source and executable may be achieved with an appropriate user interface design.

However, most procedures in database systems do contain free variables. Thus in general the conceptual division of the code and closures into pairs cannot take place, as many different executable procedures, each with its own different semantics, may derive from the same source description. It is therefore not possible at the interface level to unify the concepts of code and closure. This is illustrated in Figure 1.

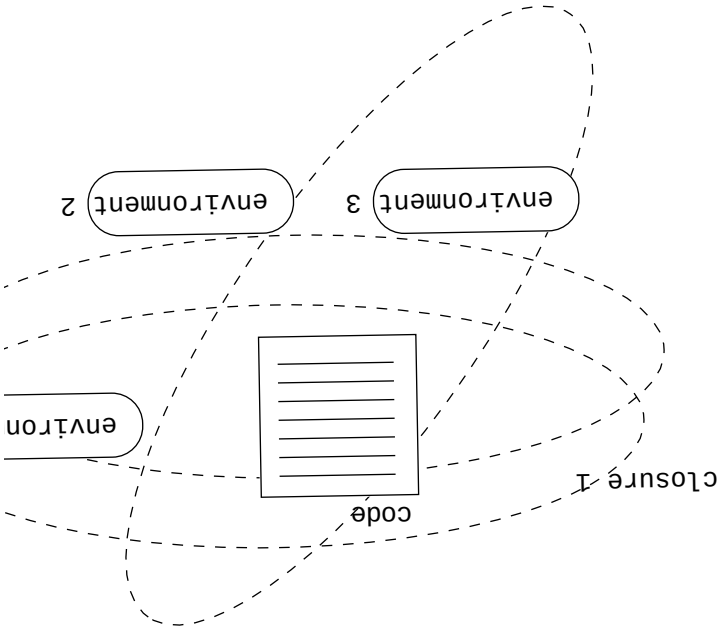

Figure 1: One to many mapping of code to closure

This problem may be overcome, however, by the use of a new paradigm for program source descriptions. This paradigm is known as hyper-programming [10], and is described in the next section. The original motivation for the design of the hyper-programming paradigm was to ease the task of source code construction; we show here how it can be further used to derive a one-to-one mapping between source and executable forms, and thus lay the foundation for their conceptual unification within a persistent programming system.

3.2 Persistent Hyper-Programming

Traditionally programs are represented as linear sequences of text. Where a program requires access to an external object during its execution, it must contain a textual description of how that object may be located. At some stage during the software process the description is resolved to establish a link to the object itself. Commonly this occurs during linking for code objects and during execution for data objects, and the environment in which the resolution takes place varies accordingly.

In such systems programs are typically constructed and stored in some long-term storage facility, such as a file system, separate from the run-time environment which disappears at the end of each program execution. By contrast, in persistent systems, programs may be constructed and stored in the same environment as that in which they are executed. This means that objects accessed by a program may already be available at the time the program is composed. In this case it is possible for a link to an object to be directly included in the program, replacing the traditional textual description. A program containing both text and links to objects is called a *hyper-program*.

Figure 2 shows a schematic view of a hyper-program. The links embedded in it are represented by some kind of non-textual token to allow them to be distinguished from the surrounding text. The first link is to a first class procedure value which when called writes a prompt to the user. The program then calls another procedure to read in a name, and then finds an address corresponding to the name. This is done by calling a procedure *lookup* to look up the address in a table package linked into the hyper-program. The address is then written out. Note that code objects (*readString*, *writeString* and *lookup*) are treated in exactly the same way as data objects (the table).

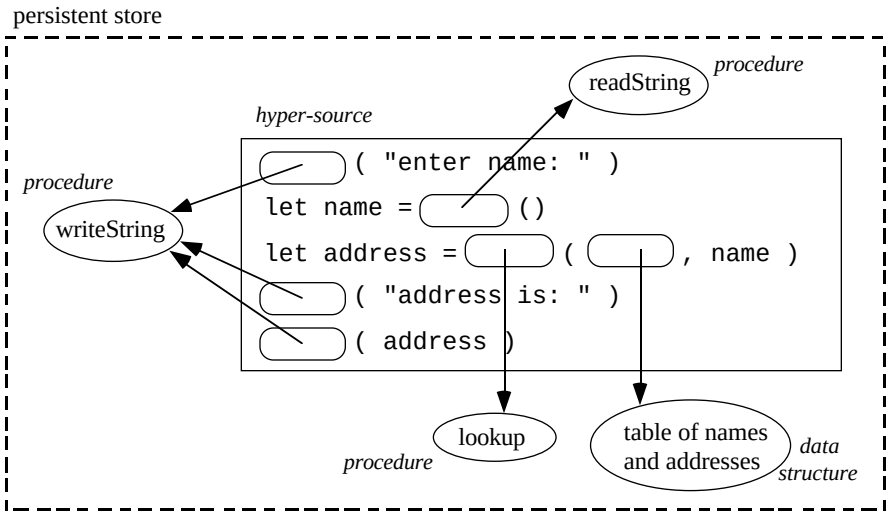

Figure 2: A hyper-program

The original motivation behind the hyper-programming paradigm was to allow programmers to incorporate uses of external data and procedures by user gesture, rather than by algebraic expression, when the external data was already present at composition time. The mechanism necessary to achieve this is composition time binding.

Non-textual tokens, referred to as hyper-links, are used within programs to denote composition time bindings. This enhanced source representation gives a way of representing non-local information in the source of a procedure; this non-local information can only be represented in textual source by the use of free variables which are resolved with respect to an evaluation environment. In Figure 2, the code contains no free variables, as all normal variable uses are superseded by hyper-links. This gives rise to the observation that the meaning of this program is independent of an evaluation environment, and is captured directly in the hyper-source.

By a simple extension of the hyper-programming paradigm it is possible to describe a hyper-source representation for any executable procedure. The concept of composition time binding is slightly widened to one of binding all free variables at closure formation time; these variables may be represented by hyper-links in exactly the same way. An example of a procedure with a free variable is shown in Figure 3. Each time the outer procedure *counterGen* is called, it creates a new integer variable *a* and returns a new procedure operating on *a*. The action of the procedure returned is to increment the variable and return its new value. All the procedures generated by *counterGen* share the same code. Their closures differ, however, since the name *a* is bound to a different location in their respective environments.

```
let counterGen = proc( → proc( → int ) )
begin
        let a := 0

        proc( → int )
        begin
                a := a + 1
                a
        end
end

let myCounter = counterGen()
let val1 = myCounter()
let val2 = myCounter()

let anotherCounter = counterGen()
let val3 = anotherCounter()
```

Figure 3: Free variable binding

The procedure *myCounter* in Figure 3, whose meaning depends critically upon its evaluation context, may be represented by the hyper-source shown in Figure 4.

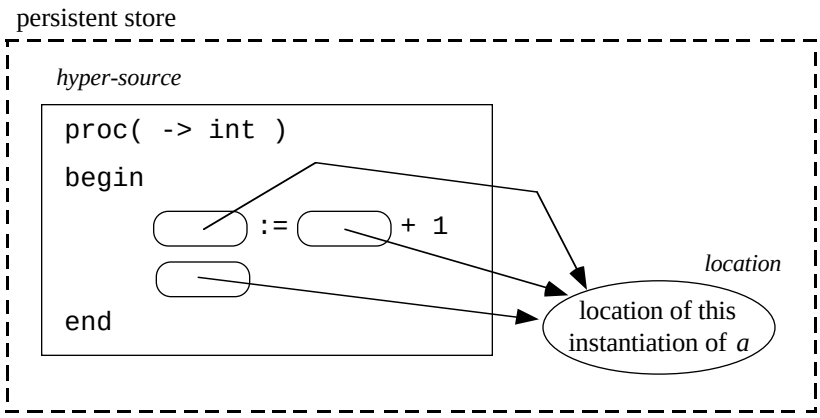

Figure 4: Corresponding hyper-source

Thus hyper-source may be used to give a source representation for any arbitrary procedure, with the property that its semantics is independent of its evaluation environment. This is quite simply achieved by representing the necessary parts of the traditional evaluation environment within the source itself, by means of hyper-links. Notice for example that successive evaluations of the procedure *counterGen* will result in different hyper-source representations, as although the textual code is the same the hyper-links are different, reflecting the different semantics. The use of hyper-source is thus able to achieve the desired one-to-one mapping between procedure source and executable forms, as illustrated in Figure 5. This clears the way to the presentation of a unified visualisation of the two representations to the programmer, leaving concepts such as compilation and linking to be matters of system efficiency rather than system building essentials.

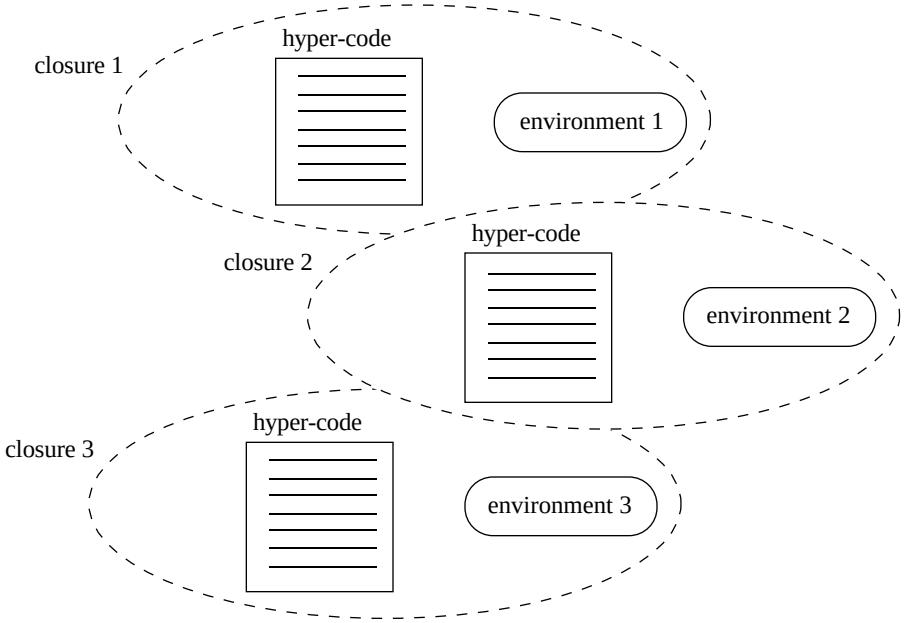

Figure 5: One-to-one mapping of hyper-code to closure

4 The Generic Browser

Values are displayed differently by the browser according to their particular type constructor. The following concepts however are upheld across all type constructors:

- Component values and locations are displayed in a uniform manner, with identical operations available. Hyper-links within procedures are deemed to be component values; thus an integer field in a structure (record) appears in a manner identical to an integer hyper-link within a procedure representation.
- Type constructors which represent abstractions over the type system are dealt with uniformly; thus a value typed as a variant which is actually a structure is displayed as a structure, with appropriate annotation to show the type widening. Thus types corresponding to value constructors, such as structure and vector, are distinguished from those representing type system abstraction, such as variants and abstract data types.
- Source code may be generated for any value displayed in the browser. If the value is a procedure, then the source "generated" will correspond to the programmer's original description of the procedure. However source will be generated automatically for any other value also, which may or may not correspond to its original derivation. The source presented is hyper-source.

- Operations on source include textual editing and “reification”, which corresponds roughly to compilation and evaluation. If the source is incorrect this results in an error; otherwise the resulting value is displayed as an anonymous value in the browser. Source representations may also be stored in some other environment if required.

4.1 General Browser Interface

In general, values are displayed in the browser in one of two formats, called *maxi* and *mini*. The mini representation is roughly equivalent to an iconic view of the value, with the maxi representation showing the full details of its construction. A maxi view will typically contain representations of its component values depicted as minis. A useful parallel may be drawn with the Macintosh operating system, where the windows displayed correspond to a type constructor representing a heterogeneous set, and the icons in these windows correspond to the mini representations of the values contained in the set.

Different styles of windows are also required, one for each data type in the language corresponding to a value constructor; as already mentioned, this includes procedures. These constructors in general contain their component icons in a rather more rigid framework than that of a heterogeneous set, but the basic principles are the same. The browser also supports the kind of user gesture associated with this interface, such as copying and moving values, and the expansion of a mini or icon representation into its own maxi or window level representation.

Another concept which is modelled at this level is that of a location, along with destructive update. Data types which support locations, such as structures, vectors and procedures, display the mini representations within a bounding box which represents the location. Destructive update is modelled by dropping another value within this box; this will only succeed if the new value has an appropriate type.

The last general concept in the browser is that of generating source. This may be done with either a maxi or mini representation; the effect is to produce a new window which contains editable hyper-source. Source may be edited and evaluated; its evaluation (if successful) results in a new anonymous value being depicted within the browser. The source produced from a value has the property that, when evaluated, the resulting value is equal to the original in every respect except for identity. Thus for any scalar value v, $evaluate(\ source(\ v\ )\ ) = v$, whereas for objects with store semantics this is not the case where the equality operator is interpreted as identity. The source generated is not a part of the conceptual value space of the browser, but is treated as a transient entity as depicted in Figure 6.

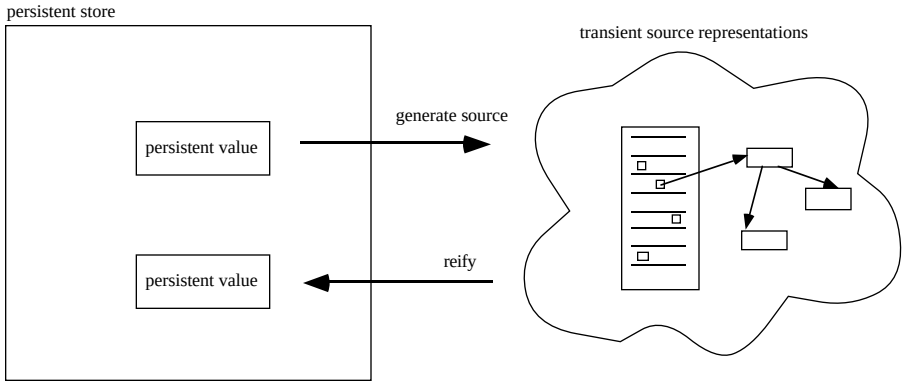

Figure 6: Persistent values and transient source

The source generated for a value is calculated only to the first level; that is, any component values—those displayed as mini representations within a maxi view—are represented as hyper-links within the source. Thus there is a very close correspondence between mini views and hyper-links, and in fact it may be reasonable in the future to merge the concepts. They are kept distinct at present to emphasise the separation between the transient source forms and the concrete maxi views of procedure values.

4.2 Browsing Values and Locations

4.2.1 Mini Representations

An example of a mini representation for an integer value is shown in Figure 7. There are two parts to the representation: the type and value areas. The type area indicates the base type or constructor; for constructors the user can double-click on the type area to pop up a more detailed representation of the type. In general the type information is displayed by means of an icon; this may be user-defined in the case of user-defined types. Our examples, however, show simple strings rather than icons for the sake of readability. Double-clicking on the value area, known as *inspecting* the value, shows more detail by replacing the mini representation with a maxi representation.

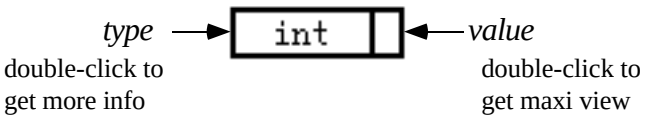

Figure 7: Mini representation

Mutable locations are shown by surrounding the mini representation with a box as shown in Figure 8:

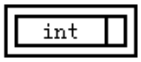

Figure 8: Mini representation of a mutable location

A new value may be assigned to the location by dragging the appropriate mini or maxi onto the box. The update succeeds only if the new value has a compatible type.

The user may select either the location or the value which it currently contains by clicking on the appropriate part of the representation. This is necessary for operations such as assigning the value to some other location or including a link to the location or value into source code. These operations will be described shortly.

4.2.2 Maxi Representations

Maxi representations of values show more information than minis and may themselves contain mini representations. Figure 9 shows both the mini representation of a structure value, and the corresponding maxi representation obtained by double-clicking on the value area of the mini. The maxi representation may be converted back to the mini by double-clicking on the title bar.

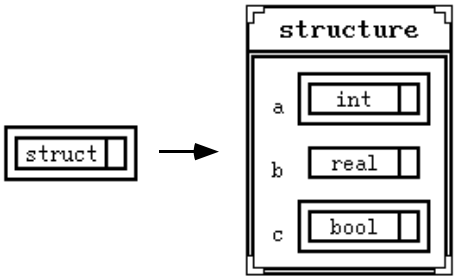

Figure 9: Mini and maxi representations of a structure

This shows a structure (record) with mutable fields *a* and *c*, and a constant field *b*. The form of a particular maxi representation depends on the type constructor involved, but the unifying theme is that all components are represented by minis within the maxi representation.

Scalar values have no internal structure; their maxi representations display the value itself in a convenient format, usually textual. An example of an integer value is shown in Figure 10:

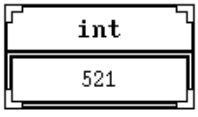

Figure 10: Maxi representation of an integer

Figure 11 shows some other examples of maxi representations, for an image, a procedure and a vector. The procedure representation shows a procedure generated by a call of *counterGen* in Figure 3. Each free variable is treated as a component in the same way as a structure field or vector location, and is represented by an embedded mini. Mutable locations can be updated by dragging new values over the box.

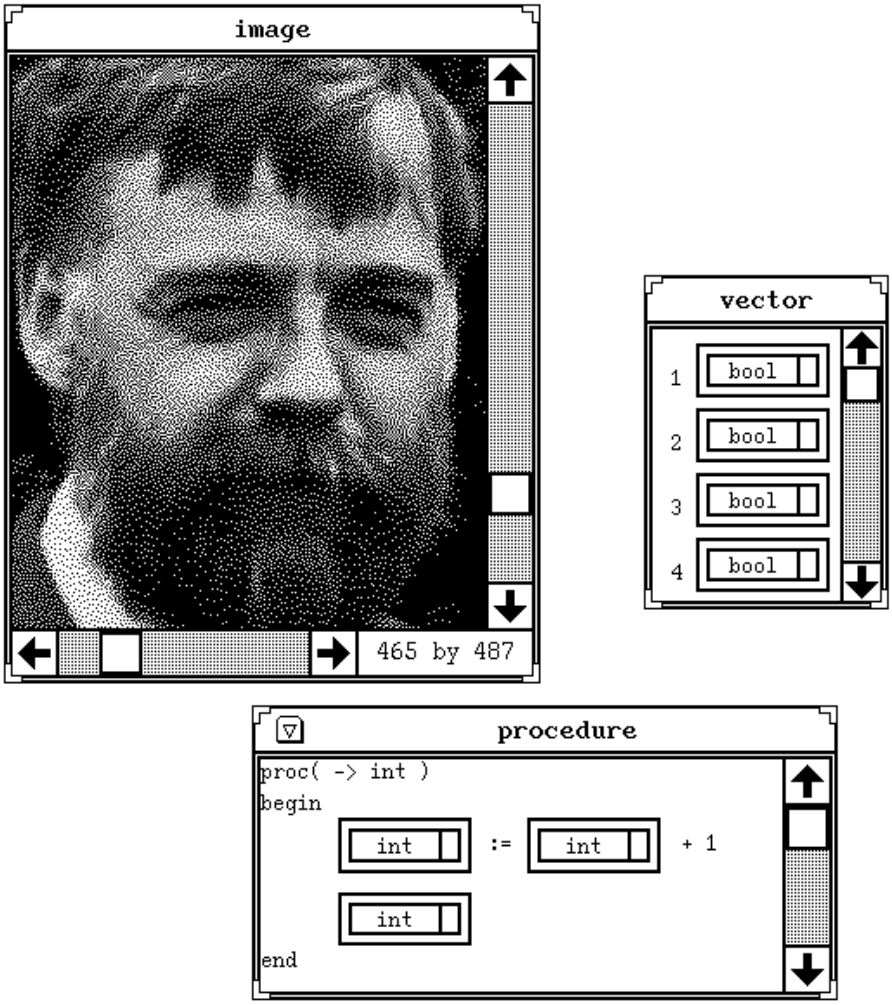

Figure 11: Maxi representations of an image, procedure and vector

4.3 Inspecting Maxi Components

A component of a maxi representation may be inspected by double-clicking on the corresponding embedded mini representation. If the mini is of a scalar type this results in the mini being expanded into a maxi in place, within the containing maxi.

For other values the maxi is displayed as a separate window and a link is drawn to it from the parent maxi. This approach reflects the treatment of equality in Napier88, in which two scalar values of the same type are equal iff they are bitwise identical, whereas two non-scalars are equal iff they have the same identity.

Figure 12 shows an example of the maxi representation of a structure and the expanded representation of one of its fields.

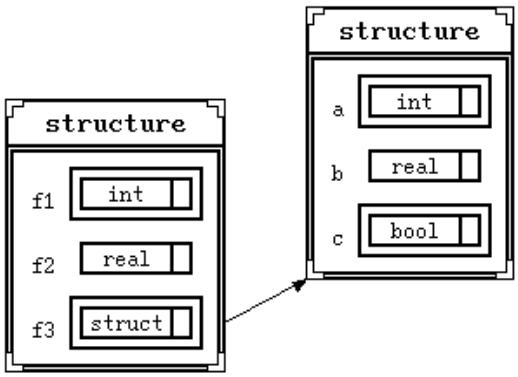

Figure 12: Inspecting a component of a maxi

4.4 Type Abstraction

Several of the Napier88 types and type constructors involve abstraction over the type of a value. These are **any** (an infinite union), variants (labelled disjoint sums) and abstract data types. The browser displays such values by annotating the mini and maxi representations with extra type information. Figure 13 shows an example of the representations of an instance of the following variant type:

rec type list **is variant** (cons : **structure**(hd : **int** ; tl : list) ; tip : **null**)

The main part of each representation shows the value itself in the normal way. Attached to the edge is an annotation describing the type abstraction through which the value is currently viewed. The example shows an instance of the *cons* branch, indicated in the maxi representation by emboldening that branch name. The user can double-click on the annotation title bar to obtain a detailed description of the type. It is also possible to select the value either as an instance of the variant type, or as an instance of the branch structure type, depending on which part of the representation is clicked on.

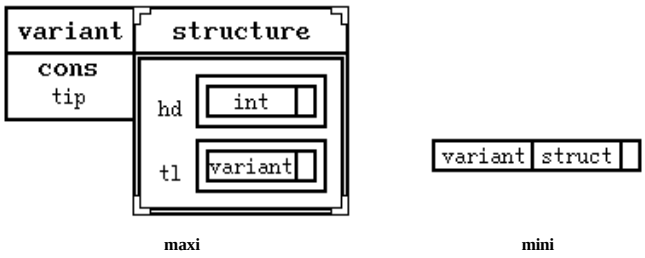

Figure 13: Annotated variant representations

A similar approach is used to display both **any**s and abstypes. In the first case the annotation simply indicates that the value is injected into **any**. For abstypes the annotation indicates that some types are abstracted over—the witness types—and the user operations on fields dependent on such types are appropriately limited.

4.5 Hyper-Source Operations

The user may request the system to generate hyper-source for a selected value. This hyper-source may then be edited, involving both normal text editing and insertion of links to values and locations. Insertion of a link is achieved by selecting the appropriate mini or maxi representation and then pressing a *link* button. This inserts a link to the value directly into the source code. Once edited the source code may be reified to give a value which is displayed as a mini or maxi by the browser. Reification involves compilation of the source and execution of the resulting code. These actions, however, are hidden from the user.

Figure 14 shows an example of the hyper-source code generated for a structure value. Components of the structure are shown as embedded links.

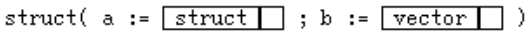

Figure 14: Hyper-source generated for a structure

If source is generated for a value and then reified again without editing, the result is a one-level copy of the original value. Thus the new value has a new identity but any components are shared with the original. It is also possible to selectively copy to any required depth by further generating, in place, source for embedded links. Figure 15 shows how the source in the previous example can be further expanded by selecting only the structure representation and generating source for it. The embedded link is replaced by the corresponding source which may in turn contain further links.

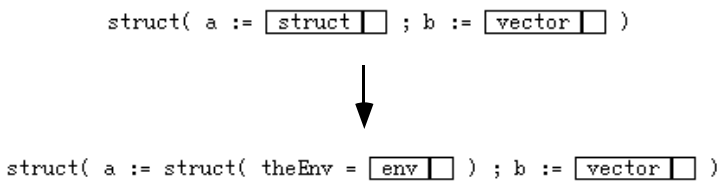

Figure 15: Selective source generation

5 Conclusions

A generic browsing methodology suitable for use with object-oriented and procedure-oriented database programming languages has been described. The new contribution of this methodology is that the browser traverses data and code in an orthogonal manner, giving a fully general interface to the database which can subsume the normal code construction mechanisms. The interface to the browser is fully generic; the main unifying features are as follows:

- Data of any type are supported in a uniform manner, including procedure closures.
- Source code is treated not as a fundamental building block within the programming system, but instead as a transient text-based view of any value.
- The artificial distinction between source and executable values within a running system is completely removed, as a consequence of using the hyper-source code representation.

The main concepts of the new browser are fully applicable to any database programming language system which treats code as data. Examples have been given in the language Napier88, and are taken from a browser which has been largely implemented. The Napier88 hyper-programming system, complete with window manager, in-store compiler, hyper-program editor and an earlier browser is available from the authors.

References

1.* Morrison R, Brown AL, Connor RCH et al. The Napier88 Reference Manual (Release 2.0). University of St Andrews Report CS/94/8, 1994

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

3. O'Brien PD, Halbert DC, Kilian MF. The Trellis Programming Environment. In: Proc. International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, 1987, pp 91-102

4.* Dearle A, Brown AL. Safe Browsing in a Strongly Typed Persistent Environment. Comp. J. 1988; 31,6:540-544

5. The LOOKS User's Manual. Altaïr , 1989

6. Bretl B, Otis A, Penney J et al. The GemStone Data Management System. In: W. Kim and F. Lochovsky (ed) Object-Oriented Concepts, Applications, and Databases. Morgan-Kaufman, 1989

7. Cooper RL. On The Utilisation of Persistent Programming Environments. Ph.D. thesis, University of Glasgow, 1990

8.* Kirby GNC, Dearle A. An Adaptive Graphical Browser for Napier88. University of St Andrews Report CS/90/16, 1990

9. Moore VS. A Hyper-Code Browsing System. University of St Andrews , 1994

10.* Kirby GNC, Connor RCH, Cutts QI, Dearle A, Farkas AM, Morrison R. Persistent Hyper-Programs. In: A. Albano and R. Morrison (ed) Persistent Object Systems, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy. Springer-Verlag, 1992, pp 86-106

*Available via *ftp* from
`ftp-fide.dcs.st-andrews.ac.uk/pub/persistence.papers`

or via *WWW* from
`http://www-fide.dcs.st-andrews.ac.uk:8080/Publications.html`