This paper should be referenced as:

Kirby, G.N.C., Cutts, Q.I., Connor, R.C.H. & Morrison, R. “The Implementation of a Hyper-Programming System”. University of St Andrews Technical Report CS/93/5 (1993).

The Implementation of a Hyper-Programming System

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

Department of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland.
{graham, quintin, richard, ron}@dcs.st-andrews.ac.uk

1 Introduction

Hyper-programming is a style of programming applicable to strongly typed persistent systems [ABC+83], in which a source program may contain direct links to language values as well as textual program constructs. It represents a form of hyper-media [EE68, YMV85, Bro86, Big88, Shn89, DMD92] applied to the program development process, in which the links may point to any typed objects. To allow this richer form of source program, the representations of the programs themselves must be held within a persistent store.

The use of hyper-programming with a persistent language gives a number of benefits including the following:

- increased ease of program composition;
- being able to perform program checking early;
- being able to enforce associations from executable programs to source programs;
- support for source representations of procedure closures; and
- availability of an increased range of linking times.

These benefits will be outlined only briefly here; they have been elaborated in greater detail in [Kir92, KCC+92]. This paper focuses on the implementation of a prototype hyper-programming system constructed for, and using, the language Napier88 [MBC+89]. Although here we report on experience with one particular language we believe that the techniques described could be applied to other persistent languages, examples of which include Amber [Car85], Galileo [ACO85], persistent Quest [MMS92], E [RC90] and STAPLE [DM90].

Section 2 explains in more detail the concept of hyper-programming and its benefits listed above; Section 3 identifies several varieties of supporting technology required to implement a hyper-programming system; Section 4 describes the user interface of the prototype system and its implementation; Section 5 describes the three different forms in which hyper-programs are represented during different phases of the program life-cycle; and Section 6 gives details of the hyper-program compiler.

2 Hyper-programming and its Benefits

An example of a hyper-program residing in a persistent store is shown in Figure 1. The hyper-program contains both text and a direct link to another data item in the persistent store. The data item is a procedure to write out strings. The direct link is a reference that cannot be accidentally or maliciously corrupted. In a strongly typed language that supports referential integrity, the link might be implemented as a pointer.

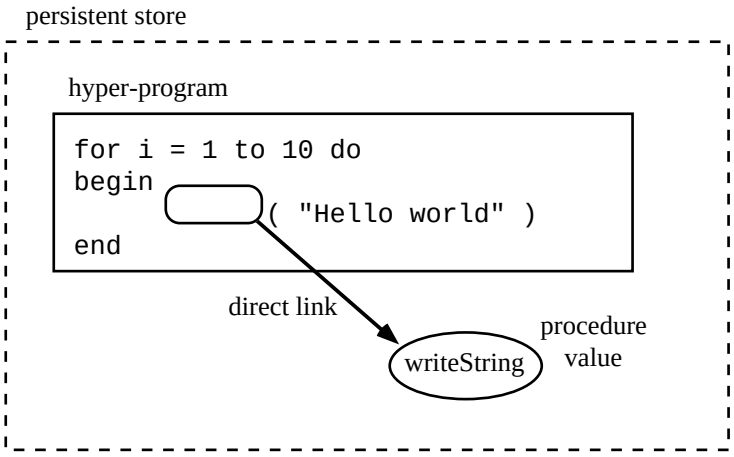

Figure 1: A hyper-program

As listed in the introduction, the use of hyper-programs as source representations offers a number of advantages over purely textual source programs.

2.1 Ease of Program Composition

The principal benefit of a hyper-programming system is a reduction in the effort required to construct programs. It allows the programmer to compose programs interactively, navigating a graphical representation of the persistent store and selecting data items to be incorporated into the programs. This removes the need to write textual specifications of the access paths for persistent data items (i.e. descriptions of how to find the data in the persistent store) referred to by a program. In conventional persistent languages such as PS-algol [PS88] and Napier88 these specifications can be verbose; in a hyper-program this verbosity is eliminated for those data items existing when the program is written. Note that the ability to link to data items at run-time is still required in the cases where data becomes available only after a program is written. As will be seen in Section 4 however, in some cases it is possible to link a store location that will later contain the data item into a hyper-program.

2.2 Early Program Checking

Hyper-programming allows some program checks to be performed earlier than in conventional systems, subsequently giving increased assurance of program correctness. This is possible because data items linked into a hyper-program are available for checking before run-time, referential integrity ensuring that the checked data remains available at run-time. Persistent data access checking is one example of checking that may be brought forward. Conventionally the consistency of the declared type of a persistent data item with its use in a program is checked at compilation-time. The existence of the data item at the declared position in the persistent store, and the equivalence of the declared type with the actual type of the data item, are then checked at run-time. The second set of checks may be eliminated when a data item is incorporated into a hyper-program: the data item is guaranteed to exist in the persistent store since it is available for incorporation in the first place; and no type needs to be declared for the data item and subsequently checked against the actual type, as the actual type is available at compilation-time.

More generally the programmer may perform arbitrary checks on data values before linking them into a hyper-program, by writing and executing other programs that compute over them. If the checks succeed, the code that performs the checking can then be omitted from the hyper-program, since the links to the original values are guaranteed to remain intact.

2.3 Enforcing Associations

In a programming system it is often desirable to maintain associations between executable programs and their corresponding source code programs, to facilitate debugging and software evolution. These associations enable the system to show the source code corresponding to the point where an error occurs in a running program, or to supply the source code for a given executable program so that it can be modified and a new version created. In file-based systems such as Pascal [Wir71] and C [KR78] these associations are maintained by convention, for example by filename suffixes as in *prog.c* and *prog.o*. With such schemes there is nothing to prevent the associations being corrupted, for example by overwriting the file *prog.c* with an inappropriate file. In an integrated persistent programming system it is possible to keep both source and executable programs within a persistent store and replace the unenforceable associations with tamper-proof, i.e. immutable, direct links from executable programs to source programs. There is then a guarantee that the appropriate source program will be accessible from a given executable program. The ability of the programming system to ensure that every executable program has its source code attached also aids software reuse, since more information about executable programs in the persistent store is available.

2.4 Increased Range of Linking Times

Persistent programming languages allow programs to be linked to persistent data at a number of different times during the program development process. These include run-time, link-time and, in a recent system, compilation-time [FDK+92]. Hyper-programming adds another possible time to this range: program composition time. Where data is linked to at composition time, safety and efficiency are increased since some checking and tracing of access paths is factored out, while flexibility is reduced since the data must be present when the hyper-program is composed. Thus a wider range of choices among the safety/efficiency/flexibility trade-offs is available.

3 Support Technology Requirements

The principal requirements for supporting a hyper-programming system are:

- a persistent store providing referential integrity;
- linguistic reflection facilities [SSS+92], including a compiler accessible from within the language and a means of denoting executable programs within the language, e.g. by first class procedure closures;
- browsing tools to display graphical representations of the data in the persistent store; and
- graphical user interface tools programmable from within the language.

A persistent store is required to contain the hyper-program representations and the data items linked into them. The store should support referential integrity, so that once a reference to a data item in it has been established, the data item will remain accessible for as long as the reference exists.

Secondly the hyper-program source representations, and the executable programs produced by compiling them, must be denotable values in the programming language. Linguistic reflective facilities are required to support the compilation process. One way to represent executable programs is as procedure closures.

A third requirement is for tools that provide the programmer with a graphical representation of the persistent store. The representation shows the values, locations and types in the store

and the links between them. The programmer can select the representations of specific data items in order to link them into hyper-programs.

4 The Graphical User Interface

Figure 2 shows how a hyper-program might appear to the programmer during editing. A more detailed description of the hyper-programming user interface is given in [Kir92].

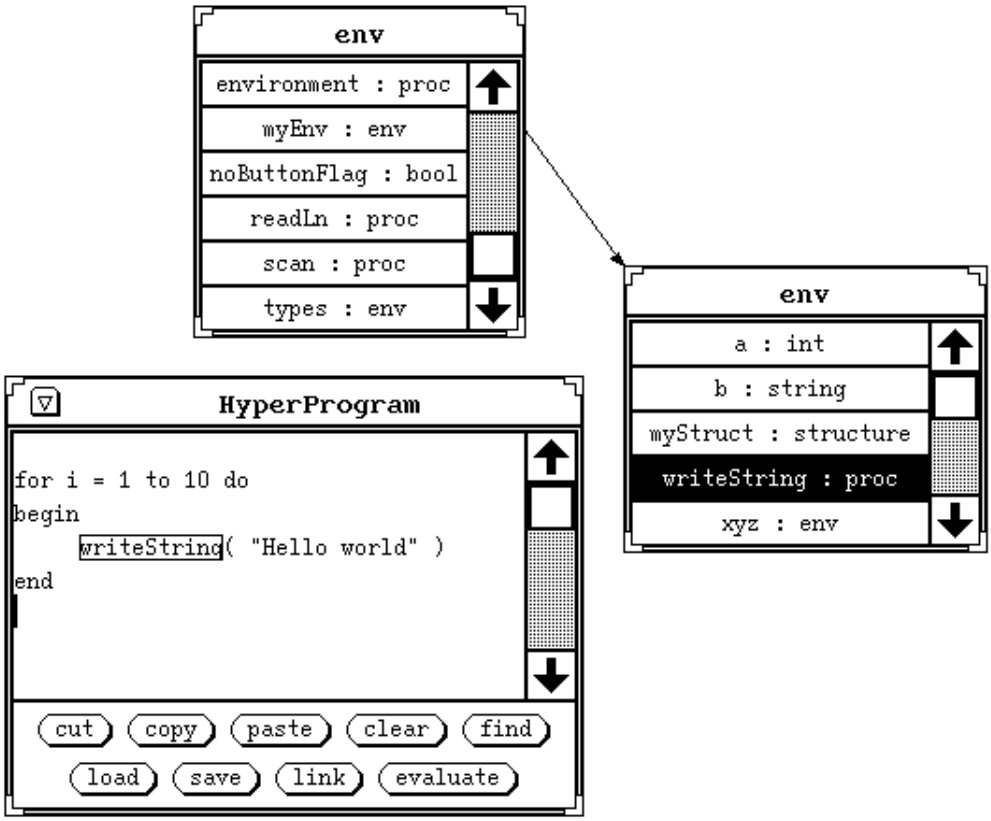

Figure 2: A hyper-program editor

The programmer composes a hyper-program by a combination of typing text into an editor window and inserting links to existing data. Browsing tools are used to display representations of the data in the persistent store and to highlight data for linking. In Figure 2 the two windows labelled *env* show the representations of Napier88 environments in the persistent store. Each environment contains a number of <name,value,type> bindings. The top-most environment in the figure contains a binding *myEnv* to the other environment; this is shown by a connecting arrow. The procedure *writeString* in this environment has been highlighted with the mouse and linked into the hyper-program where it is denoted by a light-button, by pressing the *link* button in the hyper-program editor. If the programmer presses the *writeString* button some time later, the browser will highlight the representation of the linked procedure value, redisplaying the window if necessary.

The data corresponding to a hyper-program link may be either a language value or a store location containing a value (R-value or L-value respectively). Each location may be either a location within a composite value such as a Napier88 structure or environment, or a free identifier location. This latter alternative allows hyper-programs to contain links to data in the closures of existing procedures; this will be explained further in Section 6. Figure 3 shows examples of highlighted graphical representations of, from the left, a value (an environment), a location in an environment, and a free variable location.

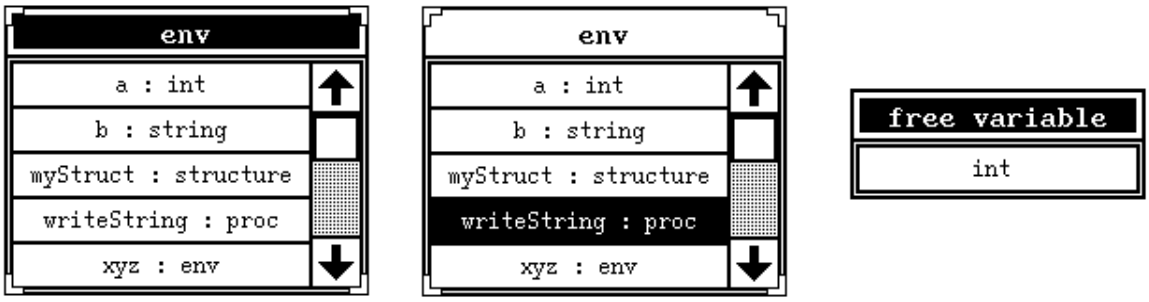

Figure 3: Graphical data representations

The light-buttons within the text in a hyper-program editor can be moved and edited like the rest of the text. The names displayed on the light-buttons can also be changed as they are not significant to the meaning of the hyper-program.

5 Representing Hyper-Programs

The hyper-programming system uses three different representation forms for hyper-programs at various stages of the program development process. The first of these, the *editing* form, is optimised for editing, including fast selection, insertion and deletion of both text and light-buttons. The second, the *export* form, is optimised for low-cost storage and is designed to occupy as little space as possible. The motivation for this is that, as will be described later, the hyper-program system automatically records a hyper-program whenever a procedure is compiled and thus the number of hyper-programs simultaneously existing is potentially large. The third representation form, the *compilation* form, is designed to allow a hyper-program compiler to be constructed from the original Napier88 compiler with minimal change.

Translation from the *editing* form to the *export* form takes place when the contents of a hyper-program editor are read using the appropriate interface procedure. This form is also passed to the hyper-program compiler for compilation. Within the hyper-program compiler the *export* form is translated to the *compilation* form before being processed.

The three representation forms will now be illustrated with reference to the example hyper-program shown in Figure 4. The hyper-program contains links to an integer, a procedure value and an environment location in the persistent store.

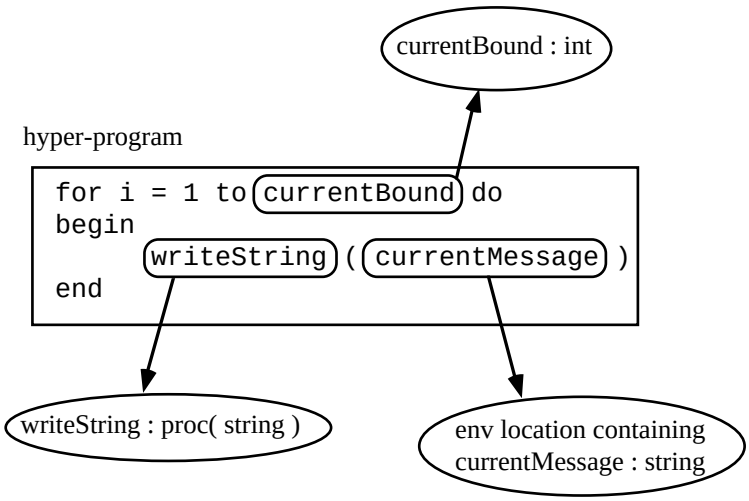

Figure 4: Example hyper-program

The *editing* form is illustrated in Figure 5 which shows the data structure describing the text, the positions of the embedded light-buttons and their associated actions. The types used are defined in the appendix.

The text is stored in a doubly linked list of strings, one for each text line. The new-line at the end of each text line is not stored as part of the string but is implicitly present between each consecutive pair of lines. Each list element also contains a line number. This allows the editor to determine the ordering for list elements efficiently.

Details of embedded light-buttons are stored in a vector. Each element contains:

- an integer index for the button;
- the text displayed on the button;
- the text positions at which it starts and finishes;
- the procedure that will be executed when the button is pressed;
- a value of type *any* that may be set by the programmer.

The light-button vector is ordered by the buttons' positions in the text. This allows the editor to distinguish efficiently between a mouse button press over a light-button, in which case its associated procedure is called, and a press over normal text, in which case the insertion point is set to the new position. The editor calculates the text position corresponding to the (x,y) position of the mouse and then uses a binary split algorithm to determine whether the text position lies between the start and finish of any of the light-buttons.

The vector displayed below the doubly linked list in Figure 5 contains three elements recording the names and positions of the light-buttons. The button indices are independent of the elements' offsets within the vector; they are used by the programmer to denote particular buttons when calling those editor interface procedures that operate on buttons.

Each element in the vector also records:

- a procedure that is called when the light-button is pressed, causing a representation of the linked data to be displayed by the browsing tools;

- a reference to the linked data comprising an instance of type *Binding* defined in Figure 6.

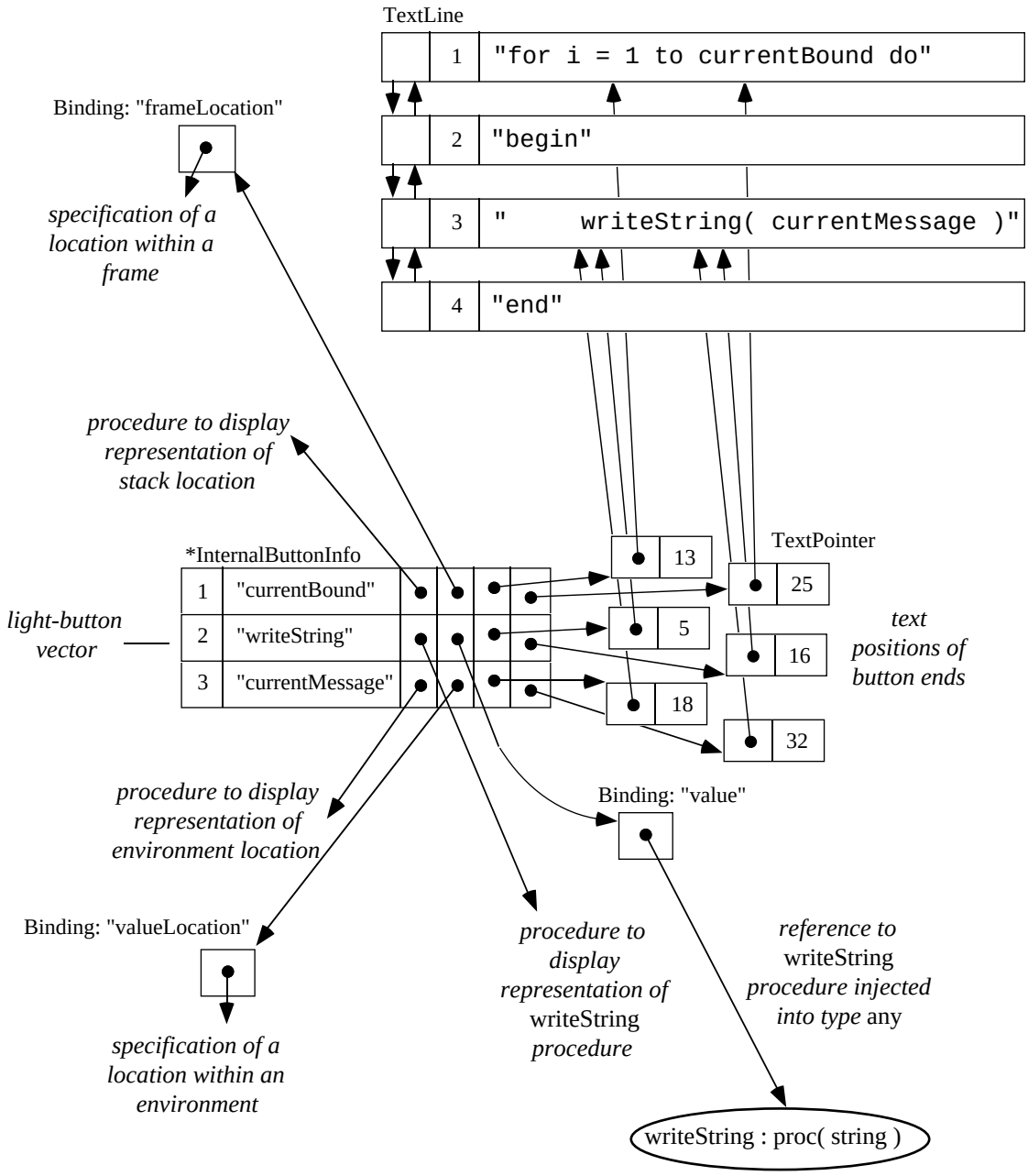

Figure 5: A hyper-program in *editing* form

The *export* form is represented by the Napier88 type *HyperSource* which is also defined in Figure 6. The definition of type *Binding* has been simplified for the purpose of explanation; the full definition is given in the appendix.

```
! *Substitution denotes a vector whose elements have type Substitution.

type HyperSource is structure(  code : string ;
                                bindings : Optional[ *Substitution[ Binding ] ] )
where

type Optional[ T ] is variant( present : T ; absent : null )

type Substitution[ T ] is structure( val : T ; codeRegion : CodeRegion )
```

```
type Binding is variant( value :          any ;  ! The value in a universal type.
                         valueLocation :  ValueLocation;
                         frameLocation :  FrameLocation )

type CodeRegion is structure( start, finish : int )      ! Specification of region of code.

type ValueLocation is …  ! Specification of a location within a composite value.

type FrameLocation is …  ! Specification of a location within a frame (uses shown later).
```

Figure 6: Type definition for *HyperSource*

The *export* form contains a string together with a vector of substitutions. Each substitution specifies a region within the string and the data, an instance of type *Binding*, to be substituted in that region. The substitution regions are specified by pairs of character offsets from the start of the string. The example hyper-program is shown in this form in Figure 7.

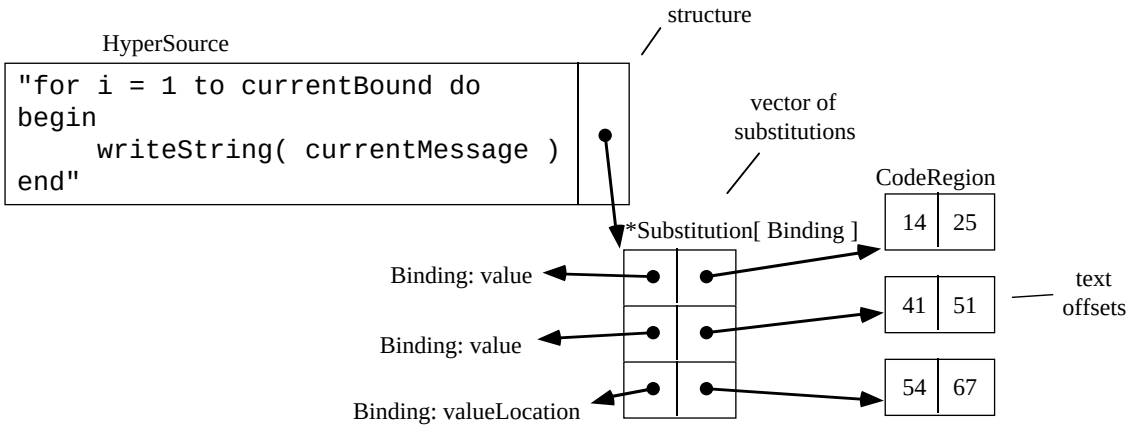

Figure 7: A hyper-program in *export* form

When the hyper-program is compiled the compiler first converts it to the *compilation* form shown in Figure 8. Each substitution region in the text string is replaced by a unique identifier of the form *uniqueId*n where *n* is an integer chosen to ensure that the identifier does not occur anywhere else in the representation. Associated with the text string is a newly created symbol table which contains an entry for each of the unique identifiers. Among other items of information, each entry contains a representation of the type of the linked data and a reference to the abstract machine representation of the data itself. The hyper-program is then compiled using the new symbol table to resolve uses of the substituted identifiers.

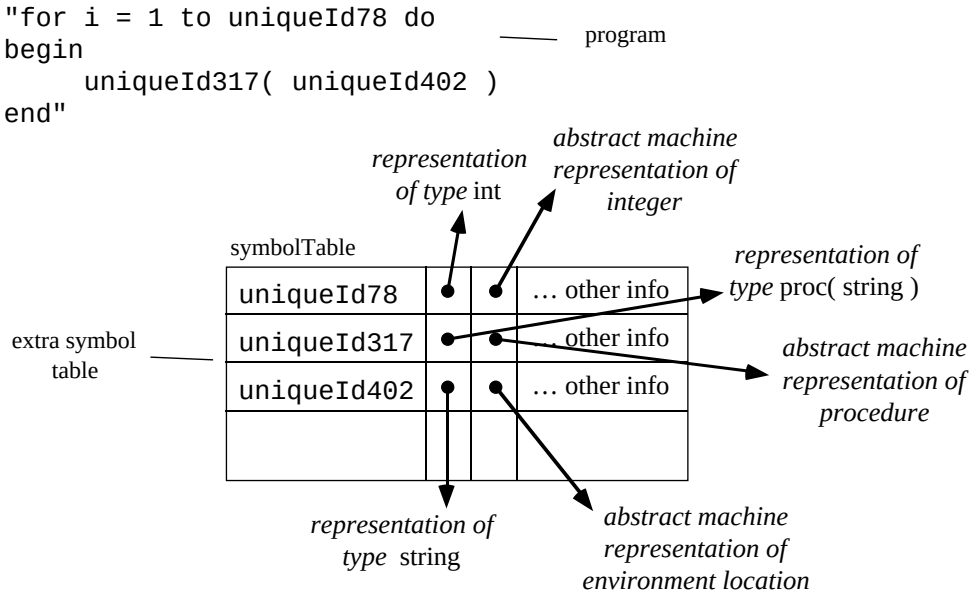

Figure 8: A hyper-program in *compilation* form

6 Compiling Hyper-Programs

6.1 Compiler Interfaces

The first Napier88 compiler was implemented by Brown, Connor, Dearle and Morrison. The hyper-programming system described here uses a modified version implemented in Napier88 by Cutts [Cut92]. The compiler is accessible by Napier88 programs through several interfaces. The simplest is shown in Figure 9:

```
compileString : proc( string → any )
```

Figure 9: Simple interface to the Napier88 compiler

This interface to the compiler is a procedure that takes a string parameter and returns an instance of the union type *any* which can then be projected to give the resulting value. There is also a more flexible interface to the compiler which allows programs to be compiled against existing values. The interface is shown in Figure 10:

```
type lValue is …                    ! structure containing info about an identifier
type symbolTable is table[ string, lValue ]

compile : proc( string, list[ symbolTable ] → any )
```

Figure 10: Flexible interface to the Napier88 compiler

This version of the compiler interface takes as parameters a string and a list of symbol tables, and returns a result of type *any*. The list contains symbol tables that form a series of extra ‘outer scopes’ during compilation. To compile a program against an existing value, the programmer constructs a new symbol table using a procedure available in the persistent store, adds the value to the symbol table and passes it in a list to the compiler. If the compiler encounters an identifier not declared within the source program it searches the extra symbol

tables and, if found, plants a reference to the corresponding value or store location in the resulting executable code.

6.2 Parsing Procedure Definitions

The second interface described above is sufficient to compile hyper-programs containing links to data in the persistent store. It is also useful to be able to treat procedure definitions as hyper-programs in their own right, replacing references to free identifiers by hyper-program links. This allows every procedure value to have a source level representation. Figure 11 illustrates a procedure definition and the hyper-program corresponding to the procedure value produced by executing the definition, in which the occurrences of a free identifier are replaced by hyper-program links to the appropriate store location:

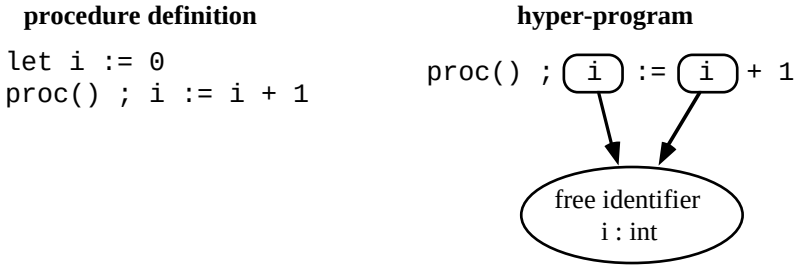

Figure 11: Procedure definition and corresponding hyper-program

In order to enforce associations from procedures to source representations, whenever the compiler reaches the end of a procedure definition it extracts the part of the source code defining the procedure and inserts a reference to it in the newly constructed code vector. The source code stored is a hyper-program, in the *export* form, in which all free identifiers are replaced by hyper-program links to store locations. The compiler records the current position within the source code as it starts to compile a procedure and again at the end, giving the textual bounds of the procedure definition. Since procedure definitions may be nested, the compiler uses a stack of procedure definition start positions to ensure correct processing.

The compiler also keeps track of hyper-program links to be inserted into the procedure source code. These occur where the source program itself contains hyper-program links, and also where a free identifier is used within a procedure definition. Free means that the identifier is declared outside the procedure definition, in an outer block. In Napier88 a procedure value is represented by a code vector and a list of the enclosing stack frames constructed during the execution of the program that created the procedure. The values of free variables are resolved with respect to these stack frames.

To determine which identifiers are free, the compiler records the lexical level of each procedure in the corresponding stack element. Whenever an identifier is encountered, the symbol table entry for that identifier is obtained. If the entry shows that the identifier was declared at a lower lexical level than that of the procedure currently being compiled, then the identifier is free. In that case a new element is added to a list contained in the top procedure stack element. The new list element contains the source text position of the identifier and a specification of its corresponding data, of type *Binding*. In the case that the identifier denotes a hyper-program link present in the source program, then the data already exists and the *Binding* contains a reference to a value or location. Alternatively, where the identifier is free, the data will not exist until run-time, and the *Binding* contains a description of the data's run-time location comprising a frame number (offset up the static chain) and a position within the frame. Each time the end of a procedure is reached the information about the current procedure definition is popped from the stack and used to produce its textual source code together with a vector of substitutions. Each substitution contains the position of an identifier and a *Binding*. The text and the substitutions together form an instance of type *HyperSource*,

a hyper-program, and a reference to this is inserted in the newly formed code vector for the procedure.

This process is illustrated in Figure 12. The source code contains two procedure definitions *p1* and *p2*, with *p2* nested inside *p1*. The lexical level before the first procedure definition is 0; at the start of *p1* it becomes 1; inside *p2* it is 2. The source character offsets of the start and finish of *p1* are denoted by *offset 1* and *offset 4*, while the corresponding offsets for *p2* are *offset 2* and *offset 3*. The identifiers *x* and *y* are declared within the program and *z* represents a hyper-program link to a value in the persistent store. The figure shows the hyper-program source representations recorded for *p1* and *p2*; note that some identifiers appear in both representations. A given identifier may appear normally in one representation and as a hyper-program link in another, as is the case for *y* in this example.

The bottom part of the figure shows the state of the procedure stack at the point that the compiler reaches the end of *p2*. The top element contains information about *p2*: its start offset, its lexical level and a list of the free identifiers used within it, *x* and *y*. Below this on the stack is information about *p1*, the free identifiers being *x*, twice, and *z*. At this point in compilation the contents of the top element are used to form the source for *p2*; the stack is then popped and later the other element is used to form the source for *p1*.

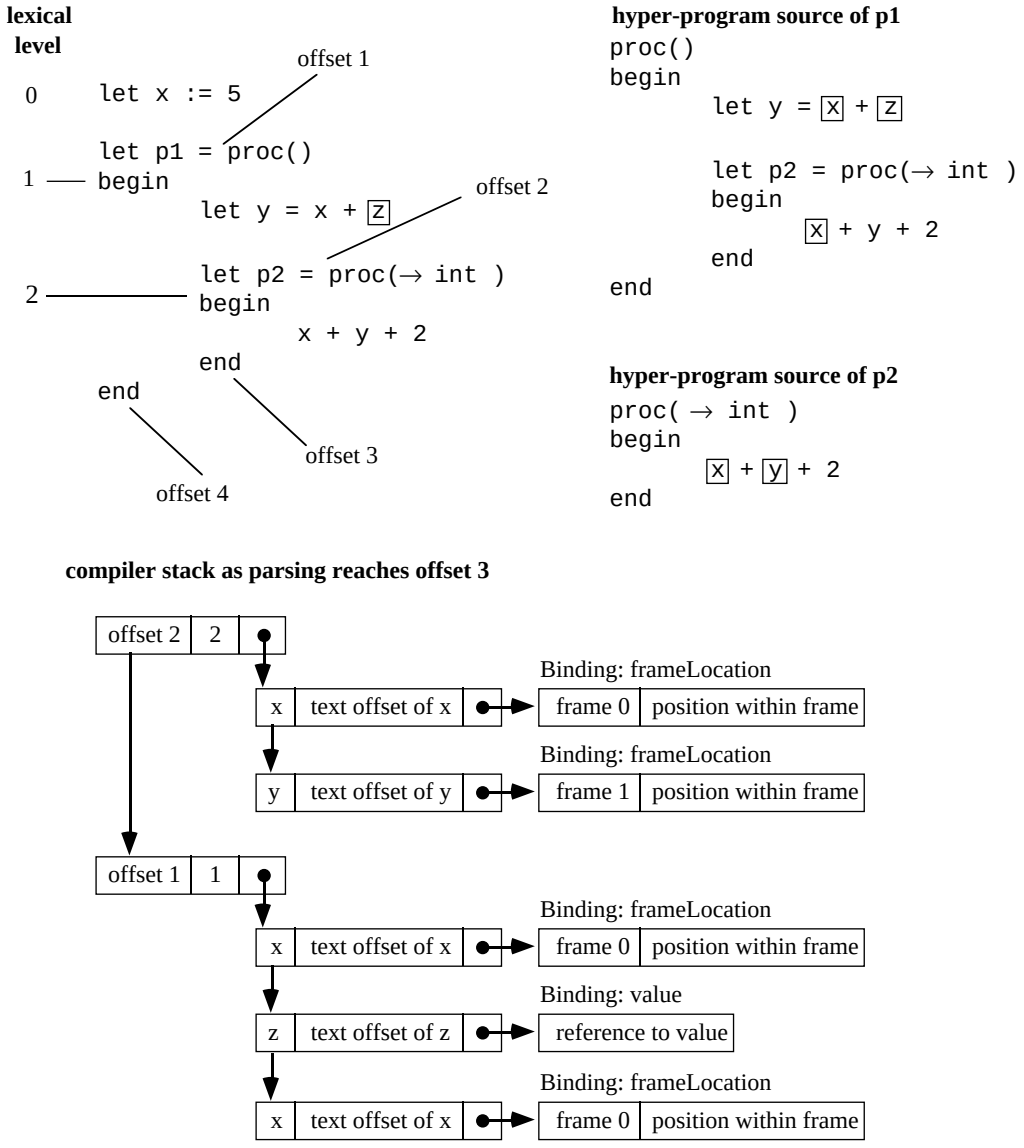

Figure 12: Constructing source representations for nested procedures

As illustrated above, the information recorded for a free identifier consists of a frame number and a position within that frame. The frame itself cannot be recorded as it does not come into existence until the compiled program is executed. When the source hyper-program of a procedure value is displayed by the browsing tools, the outer frames now exist so the hyper-program is scanned and pointers to the appropriate frames inserted. These are found by traversing the procedure's static chain to find the appropriate frame numbers. When a light-button corresponding to a free identifier is pressed the associated value is displayed by reading words from the frame, converting them to a typed value and passing the result to the browsing tools.

6.3 Compiling References to Existing Data

The mechanisms described so far allow a source program passed to the compiler to contain hyper-program links to values or locations in the persistent store. Another variation is needed to cater for the possibility that a source program may contain hyper-program links to locations within existing frames. This situation may arise when a new source program is created by combining components copied from the source programs of existing procedures with free identifiers, as illustrated in the following figures. Figure 13 shows two source programs that contain references to frames containing free identifiers. Each frame contains a pointer to the next frame in the static chain, eventually terminating in the outer-most frame. Since the two procedures have been produced by executing independently compiled programs, the two static chains are disjoint.

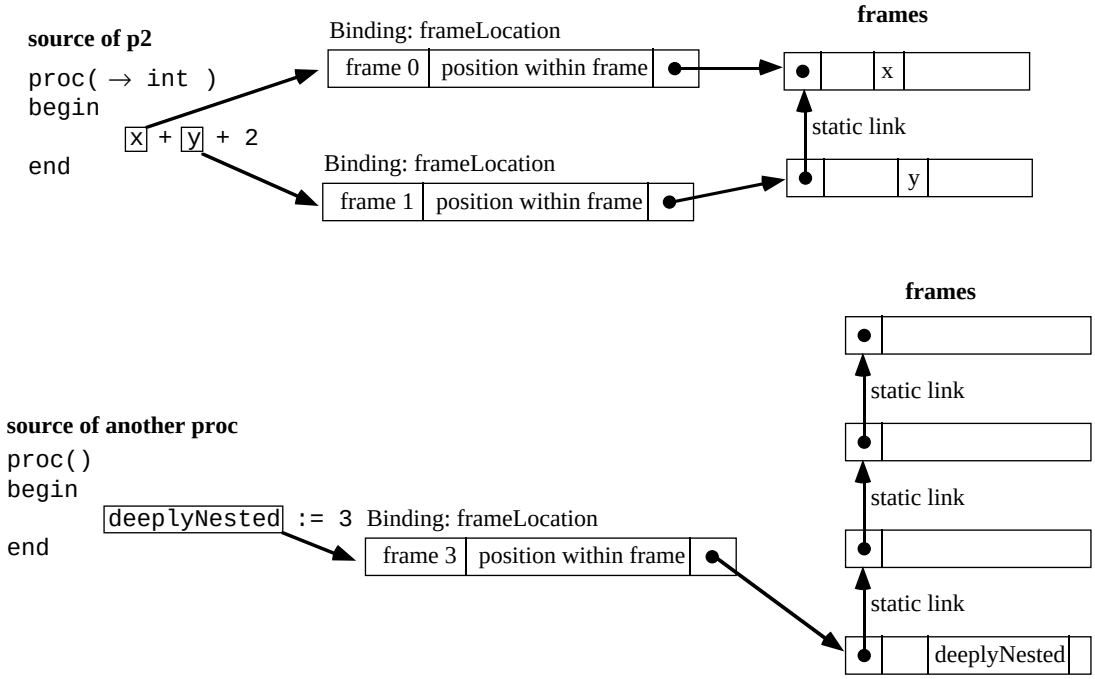

Figure 13: Procedures with disjoint static chains

Figure 14 shows a new source program constructed by copying links from both existing source programs:

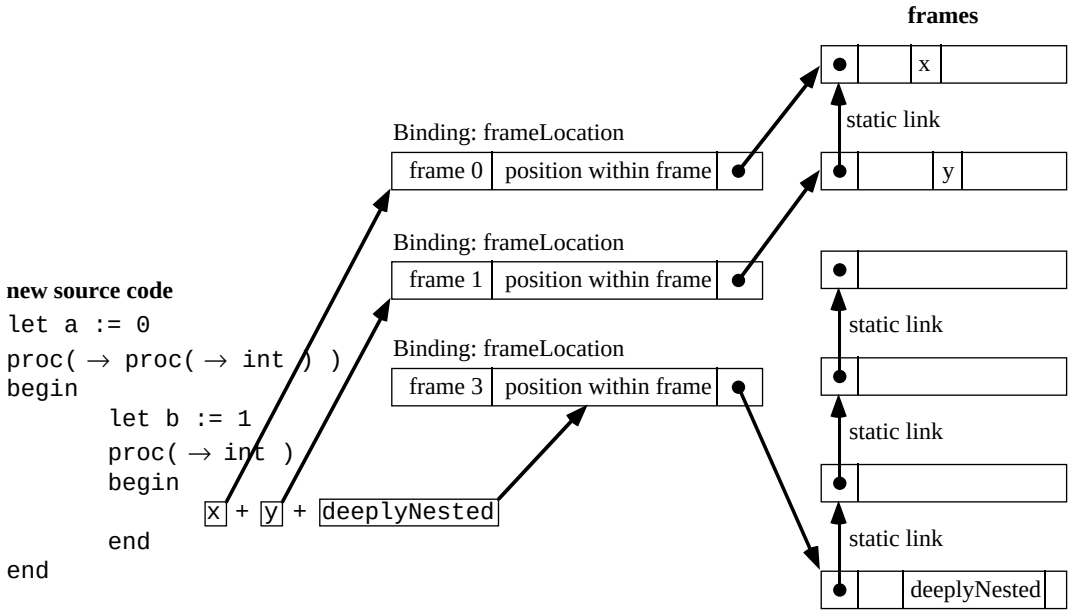

Figure 14: Program with references to existing frames

When compiling a source program containing links to identifiers in existing frames, the compiler first allocates a numbering to each of the frames. The frames are numbered consecutively from 0 and the ordering is unimportant. The compiler then modifies the frame numbers recorded with each link to reflect the new numbering scheme, and sets the lexical level at the beginning of compilation to the number of external frames. In the example shown there are 3 external frames so the lexical level at the beginning of compilation is 3. In contrast, the standard compiler always begins compilation with a lexical level of 0.

As each procedure definition is compiled the compiler generates code to build a display on entry to the procedure. When executed this code traverses the procedure's static chain, loading onto the stack a pointer to each frame in the chain. In addition it loads a pointer to each of the external frames in decreasing order of frame number. This ensures that references to external identifiers planted in the compiled code are resolved correctly at run-time.

This mechanism is illustrated in Figure 15, which shows the state of the compiler's symbol table list at the start of compilation of the body of the inner procedure in Figure 13. The first two symbol tables contain entries for the identifiers *a* and *b* declared in the enclosing blocks. Since compilation started at a lexical level of 3, this is the frame number for *a*. Another symbol table contains entries for the unique identifiers assigned to represent the external identifiers. These entries contain the frame numbers assigned to the external frames at the start of compilation.

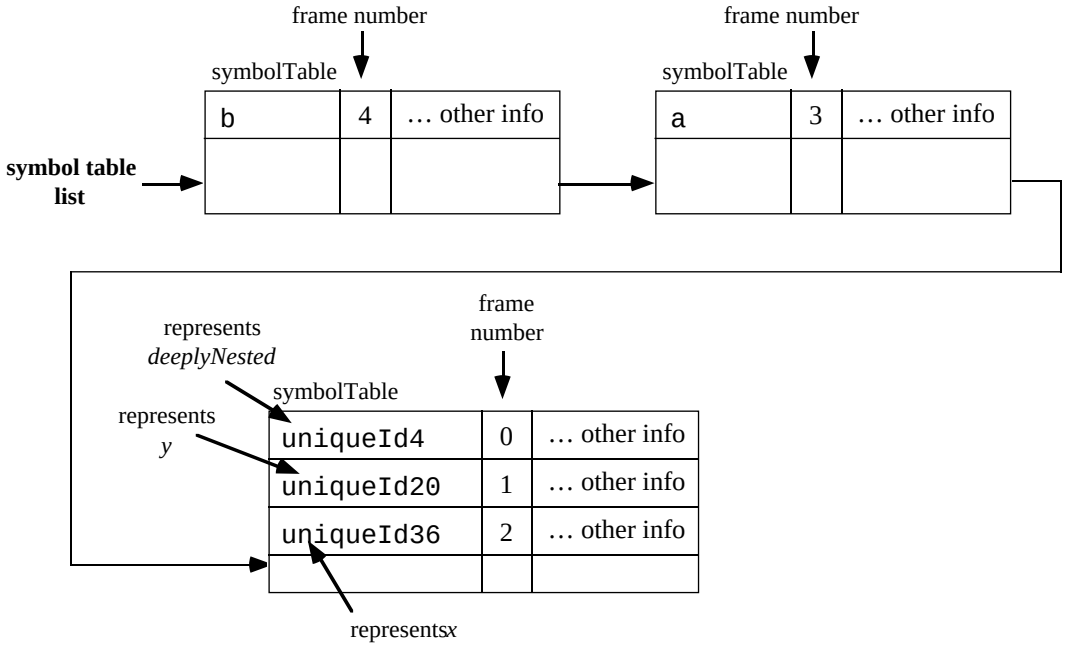

Figure 15: Symbol table list during compilation

Figure 16 shows the current frame at the start of execution of the procedure body. The frame's static link points to the frame for the enclosing block, created at run-time. The display contains pointers to this frame and to each of the external frames.

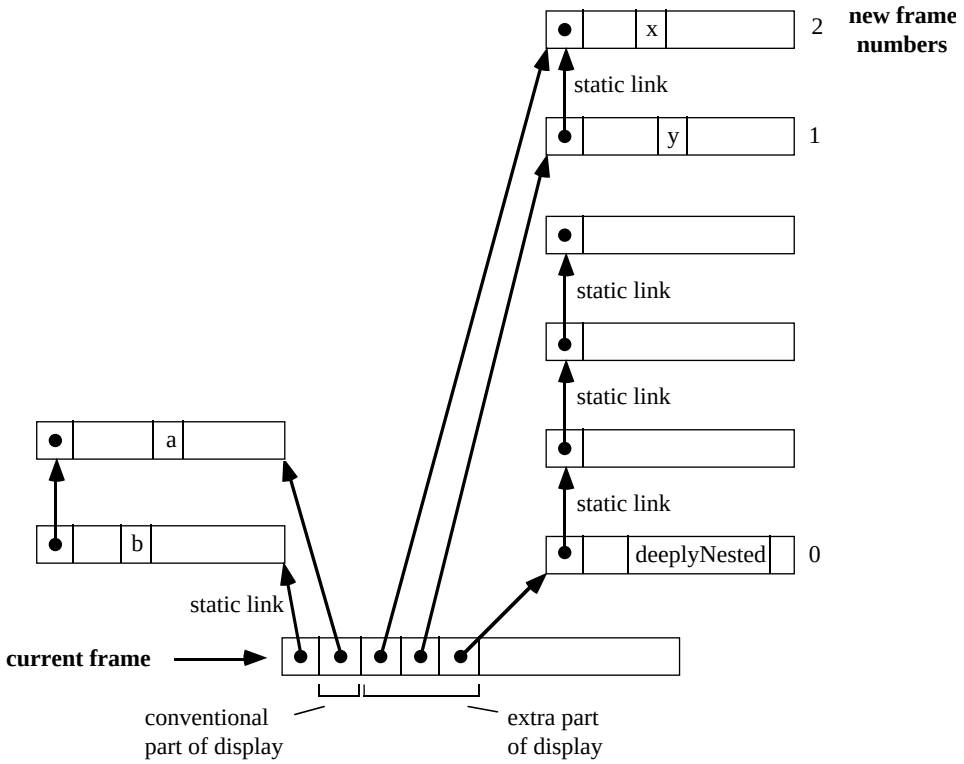

Figure 16: Current frame during execution

7 Conclusions

Hyper-programming provides a new style for linking programs with persistent data, allowing direct links to data to be incorporated into source programs held in a persistent store. The

benefits of this facility have been outlined; in particular it allows more convenient composition of programs and it allows associations between executable programs and source programs to be enforced, in contrast with traditional systems that rely on convention.

A prototype hyper-programming system for Napier88 allows the programmer to browse the data in the persistent store, to construct source programs, and to link items of data into the programs. Each data item may be a value or a location in the persistent store. A hyper-program is represented in one of several forms depending on whether it is being edited, exported outside an editor, or processed by the compiler. The Napier88 hyper-program compiler ensures that each procedure value compiled has its hyper-program source code attached to it. An outline of the techniques used in this process has been given.

Implementation Types

```
rec type TextLine is variant( cons : structure( hd : string ; index : int ;
                                                before,after : TextLine );
                              tip : null )
type TextPointer is structure( line : TextLine ; offset : int )

type InternalButtonInfo is structure( id : int ; name : string ;
                                      start,finish : TextPointer ;
                                      action : proc( int ) ; extra : any )

rec type  TypeRep is structure( label, misc, random : int ;
                                name : string ; others : Var )
&         Var is variant( none : null ; one, unique : TypeRep ; many : *TypeRep )

type EnvLocation is structure( pointer : null ; typeRep : TypeRep )

type StructLocation is structure( structValue : any ; field : string )

type VectorLocation is structure( vectorValue : any ; index : int )

type StackPos is structure( Frame,MSoffset,PSoffset : int )

type FrameLocation is structure( frame : null ; stackPos : StackPos ;
                                 typeRep : TypeRep ; envLoc : bool )

type TypeContainer is structure( typeRep : TypeRep )

type Binding is variant( value :          any;
                         envLocation :    EnvLocation;
                         structLocation : StructLocation;
                         abstypeLocation : StructLocation;
                         vectorLocation : VectorLocation;
                         frameLocation :  FrameLocation;
                         aType :          TypeContainer )
```

References

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. "An Approach to Persistent Programming". Computer Journal 26, 4 (1983) pp 360-365.

[ACO85] Albano, A., Cardelli, L. & Orsini, R. "Galileo: a Strongly Typed, Interactive Conceptual Language". ACM Transactions on Database Systems 10, 2 (1985) pp 230-260.

[Big88] Bigelow, J. “Hypertext and CASE”. IEEE Software 5, 2 (1988).

[Bro86] Brown, P.J. “Interactive documentation”. Software – Practice and Experience 16, 3 (1986) pp 291-299.

[Car85] Cardelli, L. “Amber”. AT&T Bell Labs, Murray Hill Technical Report AT7T (1985).

[Cut92] Cutts, Q.I. “Delivering the Benefits of Persistence to System Construction and Execution”. Ph.D. Thesis, University of St Andrews (1992).

[DM90] Davie, A.J.T. & McNally, D.J. “Statically Typed Applicative Persistent Language Environment (STAPLE) Reference Manual”. University of St Andrews Technical Report CS/90/14 (1990).

[DMD92] Dearle, A., Marlin, C.D. & Dart, P. “A Hyperlinked Persistent Software Development Environment”. In Proc. Hyper-Oz ’92: A Workshop on Hypertext Activities in Australia, Adelaide, Australia (1992).

[EE68] Engelbart, D.C. & English, W.K. “A research center for augmenting human intellect”. In Proc. Joint Fall Conference (1968) pp 395-409.

[FDK+92] Farkas, A.M., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R. & Connor, R.C.H. “Persistent Program Construction through Browsing and User Gesture with some Typing”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 375-394.

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 73-95.

[Kir92] Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1992).

[KR78] Kernighan, B.W. & Ritchie, D.M. **The C programming language**. Prentice-Hall, (1978).

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. University of St Andrews Technical Report PPRR-77-89 (1989).

[MMS92] Matthes, F., Müller, R. & Schmidt, J.W. “Object Stores as Servers in Persistent Programming Environments—The P-Quest Experience”. ESPRIT BRA Project 3070 FIDE Technical Report (1992).

[PS88] “PS-algol Reference Manual, 4th edition”. Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[RC90] Richardson, J.E. & Carey, M.J. “Implementing Persistence in E”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D. (ed), Springer-Verlag (1990) pp 175-199.

[Shn89] Shneiderman, B. “Reflections on Authoring, Editing, and Managing Hypertext”. In **The Society of Text**, Barrett, E. (ed) MIT Press (1989).

[SSS+92] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. & Alagic, S. “Type-Safe Linguistic Reflection: A Generator Technology”. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/49 (1992).

[Wir71] Wirth, N. “The Programming Language Pascal”. Acta Informatica 1 (1971) pp 35-63.

[YMV85] Yankelovich, N., Meyrowitz, N. & van Dam, A. “Reading and Writing the Electronic book”. IEEE Computer October (1985) pp 15-29.