This paper should be referenced as:

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).

Persistent Hyper-Programs

G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts and R. Morrison
Department of Mathematical and Computational Sciences, University of St Andrews
St Andrews, Scotland

A. Dearle and A.M. Farkas
Department of Computer Science, University of Adelaide
Adelaide, Australia

Abstract

The traditional representation of a program as a linear sequence of text forces a particular style of program construction to ensure good programming practice. Tools such as syntax directed editors, compilers, linkers and file managers are required to translate and execute these linear sequences of text. At some stage in the execution sequence the source text is checked for type correctness and its translated form linked to values in the environment. When this is performed early in the execution process confidence in the correctness of the program is raised, at the cost of some flexibility of use.

Persistent systems allow the persistent environment to participate in the program construction process. This raises the possibility of allowing the representations of source programs to include direct links to values that already exist in the environment. By analogy with hyper-text, where a piece of text contains links to other pieces of text, this source representation is called a hyper-program.

This paper outlines how hyper-programming facilities may be provided within a persistent system, discusses advantages of the technique and proposes some outstanding research areas. The advantages of hyper-programming over conventional systems include the following: it allows more convenient program composition mechanisms; it allows earlier checking; it provides more flexible linking mechanisms; it allows more succinct program representations; and it allows procedure closures to be represented at a source code level.

1 Introduction

This work is motivated by a belief that programming language systems could provide better support for the software engineering process than they do at present. Where the language is persistent, the persistent store can participate in the program construction process. The programmer composes programs interactively by navigating the persistent store and selecting data items to be incorporated into the programs. This requires direct links to the persistent data items to be represented in the program source. By analogy with hyper-text, where a piece of text contains links to other pieces of text, this source representation is called a hyper-program.

Figure 1 shows an example of a hyper-program. The hyper-program contains both text and a token that denotes a data item in the persistent store, a procedure to write out strings.

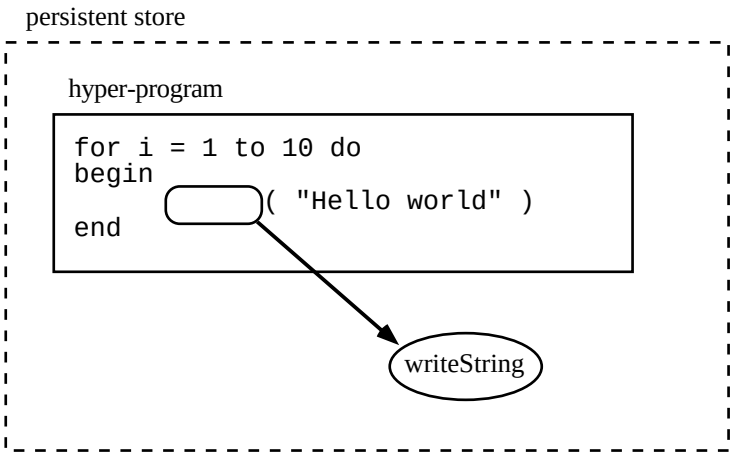

Figure 1: A hyper-program

Figure 2 shows how the hyper-program might appear to the programmer during editing. A more detailed description of a hyper-programming user interface can be found in [1].

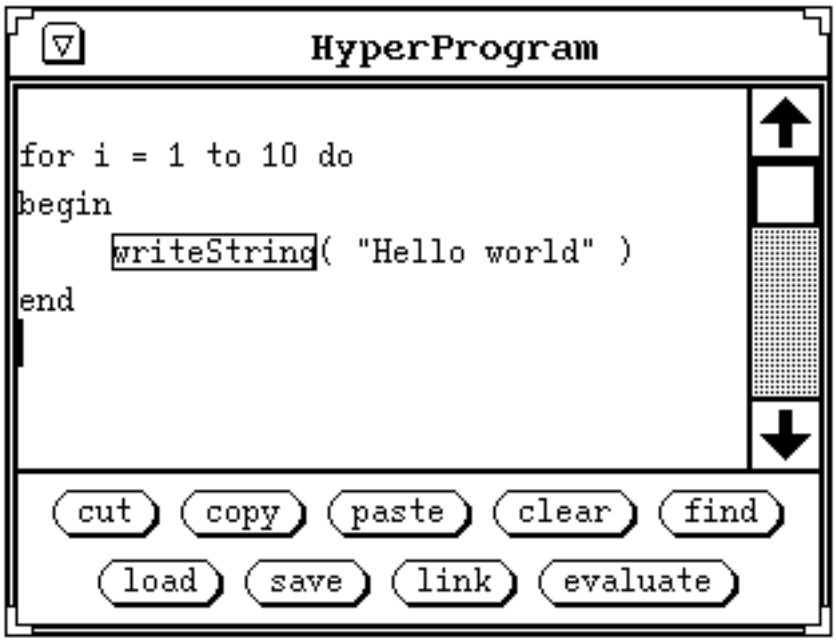

Figure 2: A hyper-program editor

The references to values are linked into the hyper-program by selecting each value with a store browsing tool and then pressing the *link* button. The system asks the programmer whether to link the program to the value itself or to the store location that currently contains the value. The editor then inserts a token at the current text position, represented by a light-button. The programmer can examine a value in a hyper-program by pressing the appropriate button in the text, which causes the browsing tool to display a representation of the value.

The benefits of hyper-programming include:

- increased ease of program composition;
- being able to perform program checking early;

- being able to enforce associations from executable programs to source programs;
- availability of an increased range of linking times;
- reduced program verbosity; and
- support for source representations of procedure closures.

The principal requirement for supporting a hyper-programming system is a persistent store to contain the program representations and the data items corresponding to the tokens in the programs. The assumption is made here that the store is stable and that it supports referential integrity. One consequence of this is that once a reference to a data item in the store has been established, the data item will remain accessible for as long as the reference exists.

Secondly, all hyper-program representations, both source and executable, must consist of denotable values within the persistent programming language environment. A further consequence is that the compilation process itself must also be supported within this same environment. One mechanism particularly well suited to realise this is known as type-safe linguistic reflection, as described in [2].

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 persistent store and the links between them. The programmer can point to the representations of specific data items and obtain tokens for them to be incorporated into hyper-programs.

To be useful in practice a hyper-programming system will also have to support additional facilities for 'programming in the large', that is, building large applications from smaller components. These include facilities for controlling the sharing of components between applications, for limiting the visibility of some components for protection reasons, and for imposing a degree of partitioning on the persistent store to aid intellectual manageability and execution efficiency. A model to support these facilities, the *hyper-world* model, is proposed.

2 Motivations and benefits

First some terms used in the following discussion will be defined:

data item: a value, or a location containing a value, in the persistent store;

access path: a description of the position of a data item relative to the root of the persistent store;

access specification: the access path of a data item together with a description of its expected type.

The principal benefits of hyper-programming are now described in more detail.

2.1 Program composition

The primary motivation for providing a hyper-programming system is to allow the programmer to compose programs interactively, navigating the persistent store and selecting data items to be incorporated into the programs. This removes the need to write access specifications for persistent data items that are accessed by a program.

Existing languages that allow a program to link to persistent data items at any time during its execution, such as PS-algol [3] and Napier88 [4], require it to contain code to specify the access path and type for each data item. The access path defines how the data is found by following a particular route through the persistent store starting from a root of persistence. The type specifies the expected type of the data at that store position. When a program is compiled the compiler checks that subsequent use of the data is compatible with its expected type. When the program is executed the run-time system checks that the data is present at the declared position and that it does have the expected type.

This mechanism gives flexibility because a program can link to data in the store at any time during its execution. However in many cases the programmer knows that a particular data item is present in the store at the time the program is written. Although the programming system could obtain all the information in the access specification by inspecting the data item at that time, the programmer must still write the access specification.

In a hyper-programming system the programmer has the option of linking existing data items into a program by pointing to graphical representations rather than writing access specifications. 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.

2.2 Early checking

Hyper-programming can provide improved safety in several ways. One of these is that it allows some program checks to be performed earlier than normal, subsequently giving increased assurance of program correctness. This is possible because data items accessed by a program may be available for checking before run-time. Referential integrity then ensures that the checked data remains available at run-time.

Checking can be performed at several stages in the program development process in existing systems. The principal opportunities are at compilation-time when a program is translated into an executable program, and at run-time when the executable program is executed. Categories of checking include checking programs for syntactic correctness and type consistency, and checking persistent data access. Usually the program checks are performed at compilation-time, although in some syntax directed programming systems [5] type consistency is verified as a program is constructed.

2.2.1 Checking persistent data access

In conventional strongly typed persistent systems a program contains an access specification for each persistent data item used. These access specifications are checked at run-time: at that time the system verifies that each data item is present in the store, with the previously declared access path and type. This is illustrated in Figure 3:

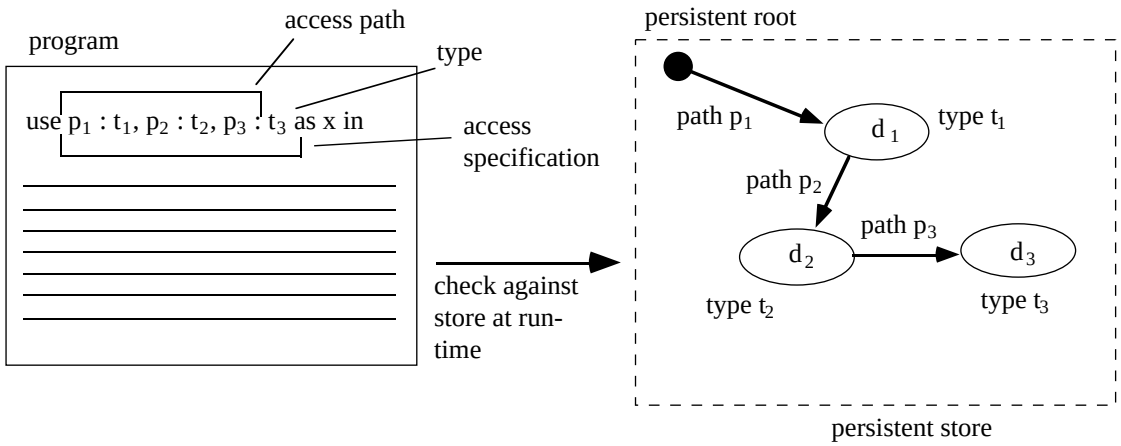

Figure 3: Access specification with run-time checking

In the program the identifier *x* is introduced to denote the data item obtained by traversing the access path $p_1 : t_1, p_2 : t_2, p_3$. In the diagram this data item is labelled d_3. The type of *x* is declared to be t_3. Each component of the path—p_1, p_2 and p_3— is a fragment of code that defines a route between two data items. p_1 is first applied to the persistent root to give data item d_1, then p_2 is applied to d_1 to give d_2, and finally p_3 to d_2 to give d_3. The types of the intermediate data items, t_1 and t_2, also form part of the access path. Note that there may be other routes to d_3 apart from the one shown. At compilation-time the system checks that the access specification is consistent with the rest of the program. At run-time it checks that the access specification is valid with respect to the current state of the store, i.e., that d_3 can be accessed along the given path at that time, and that it does have the declared type t_3.

A program execution will fail if the store does not contain a route to a data item corresponding to the access path specified in the program. Thus even if it is known at the time of writing that a particular program will execute correctly, it cannot be predicted when it may fail on some future execution.

The use of hyper-programs as source representations allows the checking of access specifications to be performed before run-time. Each token embedded in a hyper-program denotes a data item that exists in the store at the time the hyper-program is composed. The process of checking the access path is moved from run-time to program composition time. The access path is established incrementally as the programmer manipulates the graphical representations of the data in the store to locate the required data item. Once the path has been established the data item at the end of it is linked into the hyper-program and the path need not be followed again at execution time. This is illustrated in Figure 4. The hyper-program will be unaffected if the access path is then removed. This might occur, for example, due to the link from d_2 to d_3 being overwritten by a link to some other data item.

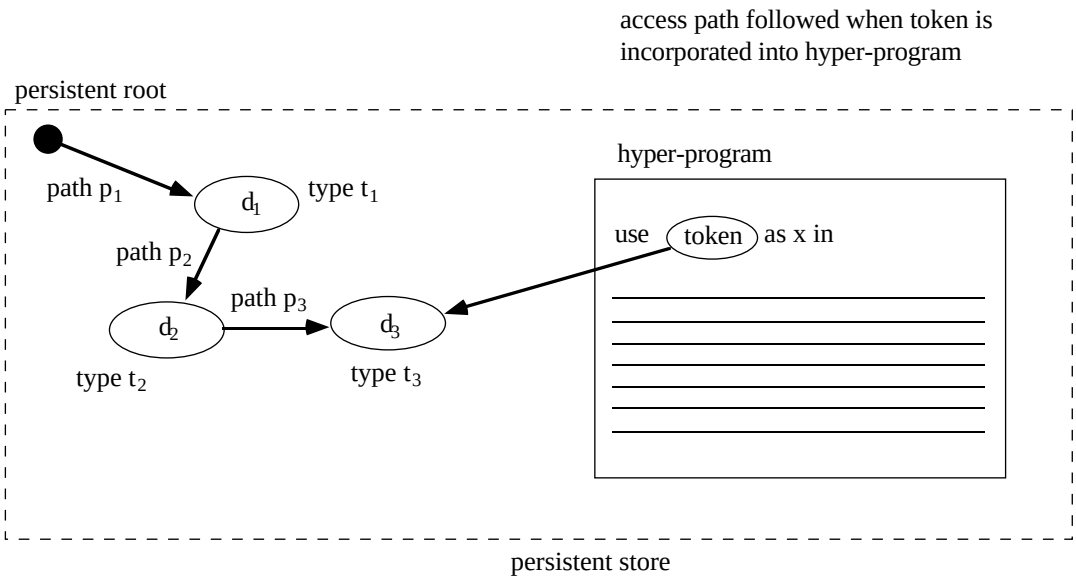

Figure 4: Access path with hyper-program

The access path part of the access specification is established during hyper-program composition. The other part, the type specification of the data item, is checked when the type consistency of the hyper-program is verified at or before compilation-time. The system checks that the type of the data item denoted by the token is compatible with the use of the token in the program.

Creating direct links from a hyper-program to values in the store, with the attendant safety benefits described above, is only applicable where values are present in the store at hyper-program composition time. Added flexibility can be gained by using tokens to denote mutable locations in the store. Linking a location into a hyper-program involves the same processes as for linking a value, with the difference that the value associated with the token changes when the location is updated. Updates to the location may occur at any time after the composition of the hyper-program. Strong typing ensures that the type of any value assigned to a location is compatible with the type of its original contents. This allows the type checking of persistent locations to be performed at compilation-time. The values in locations associated with the tokens in a hyper-program can vary but their types will always remain compatible. Where a token denotes a location, that location is linked directly into the executable program produced from the hyper-program, so that updates to the location also affect the executable program.

2.2.2 Other kinds of checking

Language systems also perform other kinds of checking at run-time, some of which can be performed earlier in a hyper-programming system. An example of this is dependent type checking.

A dependent type is a type that depends on a value. In general this requires dynamic type checking. To determine whether two dependent types are compatible, the language's type checker takes account of the associated values as well as their structure. An example of a dependent type is the generic type *map* [6], instances of which are associations between sets of values. The type of a particular map is dependent on the identity of the procedure which defines equality over the key set. Because of this it is not generally possible to type-check at compilation-time a program that contains map operations, as the map values themselves must be tested.

In a hyper-programming system the value on which a dependent type depends may be linked directly into a program, and may thus be available for checking at compilation-time. This makes it possible for the system to check operations on dependent types at compilation-time rather than planting code in the executable program to perform the checking at run-time. The system may also provide tools that allow the programmer to verify the type compatibility of selected values before they are linked into the hyper-program. Transmission of the results of such checks to the compilation system is a topic for future research.

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 main hyper-program, since the links to the original values are guaranteed to remain intact.

2.3 Source code control

2.3.1 Relationships among program forms

Safety can also be improved with respect to the relationships between executable programs and source programs. In a programming system it is often desirable to maintain links between executable programs and their corresponding source code programs, to facilitate debugging and software evolution. These links 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 existing systems these links operate by conventions and can be corrupted by programmer actions that do not conform to those conventions. Given a language that supports executable programs as first class values—for example, procedures—a hyper-programming system can enforce links from executable code to source code. To illustrate this, the relationships between these different forms of code and other data values will be described, first in general and then with particular reference to file-based systems, persistent systems and finally hyper-programming systems.

Application development involves a number of activities including the following:

- constructing source code programs;
- compiling source code to give intermediate programs;
- linking intermediate programs to give executable programs;
- linking existing data items into executable programs; and
- executing linked programs in a run-time environment.

The software entities involved in these activities are:

- source programs;
- intermediate programs—these are not executable since the code in them contains unresolved references to other programs;
- executable programs—these can be executed directly; and
- data items that are manipulated during execution.

Language systems support several varieties of relationships between the software entities listed above. These are *causations*, *associations* and *direct links*.

Causations are one-way ‘cause and effect’ relationships. A causation from an entity *A* to another entity *B* is a relationship mediated by some process having *A* as input and *B* as output. This means that a change to *A* results in a corresponding but indirect change to *B*. An example of a causation is the relationship between a source program and the corresponding compiled version, mediated by the compiler which takes the source program as input and produces a compiled version. A modification to the source program causes a corresponding change in the compiled program but only after the process of compilation.

Associations are general relationships between entities. An example is an association between an executable program and the corresponding source program, maintained by a source level debugging system. This information is not intrinsic to the associated entities themselves but is maintained by an external mechanism. In general the accuracy of associations depends on adherence to conventions: if changes to the entities are made outside the control of the external mechanism the associations may become invalid. In the example the source program could be updated without notifying the debugging system, in which case its association with the executable program would become invalid.

Direct links are references between entities in the run-time environment. A direct link from an entity *A* to another entity *B* exists if a change to *B* results in a corresponding and immediate change to *A*. This could be implemented by storing the address of *B* inside *A*. The language systems considered here support identity, that is, a reference to a given entity is guaranteed to remain valid and to refer to the same entity for as long as the reference exists. Thus a direct link from *A* to *B* always remains valid regardless of the operations performed on *B*. A change to *B* has an immediate effect on *A* without the need for any intermediate process.

2.3.2 Languages with external storage systems

In languages such as Pascal [7], Ada [8] and C [9], the persistent data, that which survives for longer than the program execution that creates it, is manipulated differently from the transient data. It is held in a storage system, separate from the run-time environment, with which programs communicate through an interface. An example is the Unix file system [10].

The program entities listed earlier—source programs, intermediate programs and executable programs—all reside in the external storage system. Source programs are compiled to produce intermediate programs. Where necessary a linker is then used to link in existing intermediate and executable programs from a program library. This linking involves combining the intermediate program with copies of the library programs to produce a new executable program. At run-time the resulting executable program is itself copied into the data space of a run-time environment and evaluated in that context. The running program may create new data items (values and locations) with direct links between them. It may also access existing data in the external storage system. The run-time environment disappears at the end of execution, along with any new data items created in it.

The relationships are illustrated pictorially in Figure 5. Here solid rectangles represent source programs, rounded rectangles represent intermediate programs, diamonds represent executable programs and ellipses represent data items that can be denoted in the programming language.

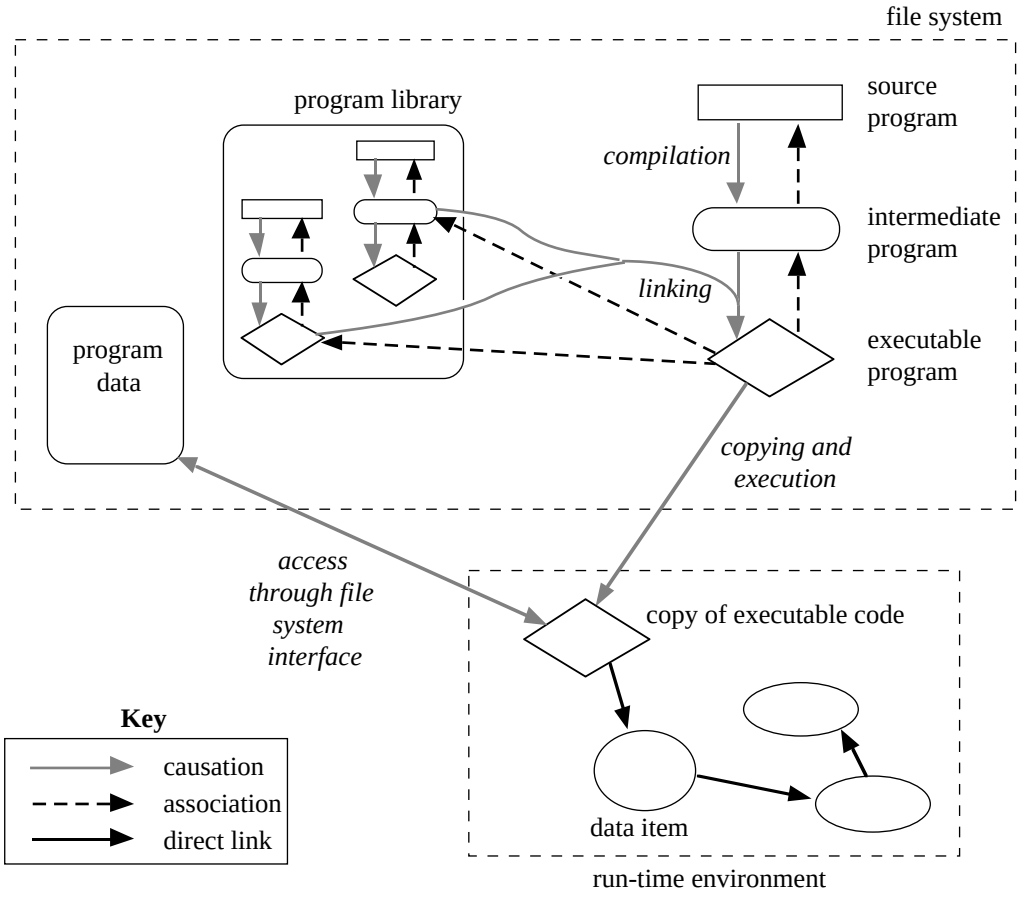

Figure 5: Relationships in a file-based system

2.3.3 Persistent languages

Persistent languages that support first class procedures are now considered. Examples of these are PS-algol, Napier88, Galileo [11, 12], P-Quest [13, 14] and STAPLE [15]. The model of persistence in these languages is persistence through reachability [16]: this means that a data item will persist at the end of a program's execution if and only if it is reachable from one or more persistent roots.

In these languages executable programs can be represented as first class procedures or functions and can thus be stored in a persistent store rather than a file system. Since each executable program is a language value it can contain direct links to other data items, and other values can contain direct links to it. A separate program library is not necessary since direct links to other executable programs in the store can be incorporated into an executable program when it is formed. Programming techniques to achieve the effects of incremental linking in this way are described in [17-20]. As executable programs are values, incremental linking of code and incremental loading of data reduce to the same problem and are handled by the run-time system.

Note that although the languages listed above use procedure closures to represent executable programs this is not essential to the schemes described in this section. All that is required is some mechanism to denote executable programs as values in the programming language.

The persistent store may subsume the functions of the file system, or the persistent store and file system may be used together. Figure 6 shows the relationships in a hybrid system in which source programs are kept in the file system and executable programs in

the store. Here the program library contains only source programs; the corresponding executable programs reside in the store. The combined ellipses and diamonds in the diagram represent these procedure values. As the linking process can be achieved without a separate linker, no intermediate programs are required.

The figure shows causations and associations between source programs and executable programs as before. There is also a causation from the main executable program e_1 to the data item v_1 which is created by execution of that program. Data item v_1 contains a direct link to data item v_2, as does e_1, which also contains direct links to other executable programs; these direct links replace the associations between executable programs and library programs shown in Figure 5.

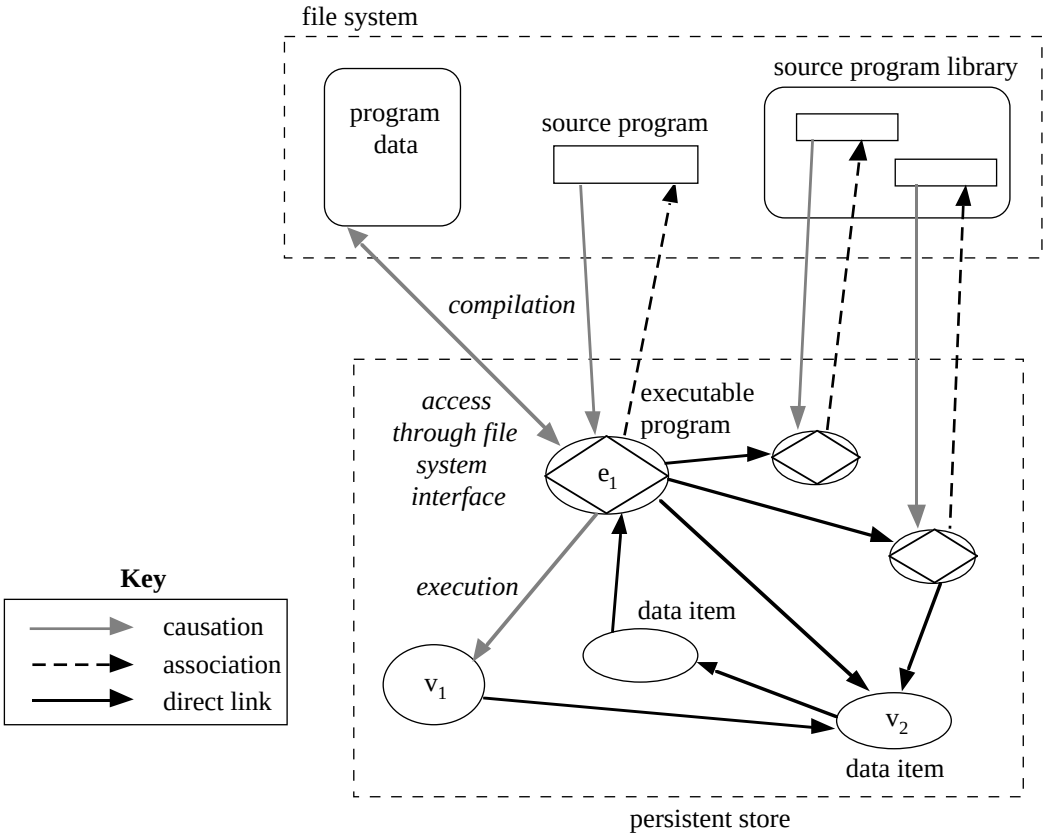

Figure 6: Relationships in a hybrid persistent / file-based system

Figure 7 shows the relationships in a persistent system where all components and data reside in the persistent store. The combined ellipses and rectangles represent source programs that are denotable values in the programming language. These values may be, for example, text strings or abstract syntax trees.

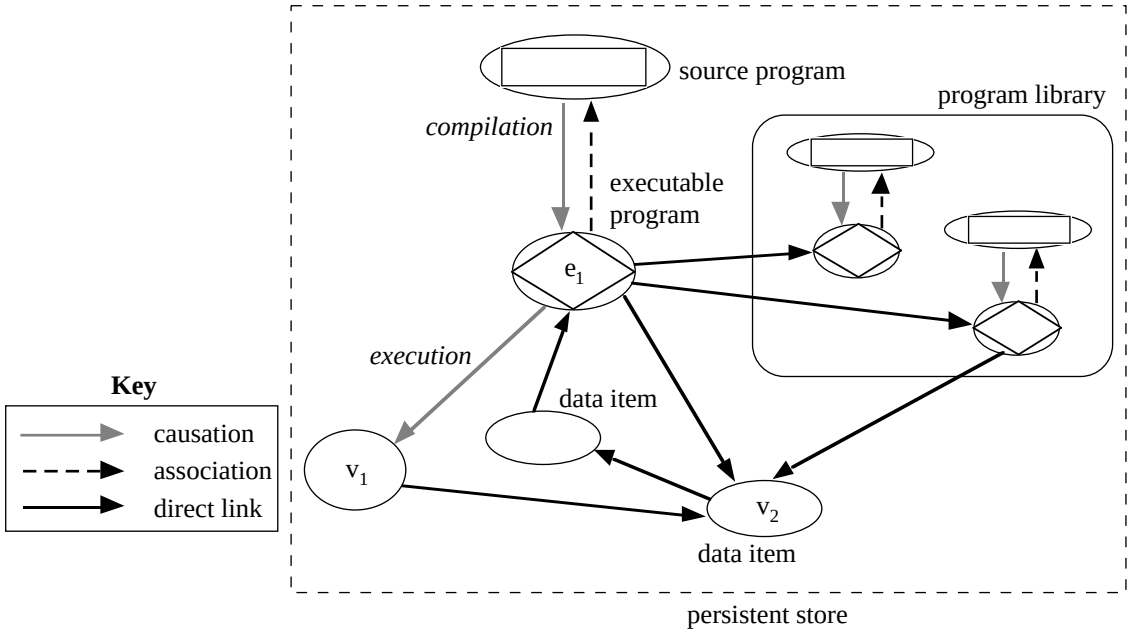

Figure 7: Relationships in a persistent system

Both schemes shown have the advantage that executable programs are associated with the others that they use by direct links. Once established these links are guaranteed to remain in place. In contrast, in a non-persistent system the integrity of the associations between executable programs that reside in the external storage system depends on the programmer following certain conventions. For example the deletion of an executable program from the program library might break these conventions.

The scheme shown in Figure 7 has the further advantage that the source programs, being in the persistent store, are brought under control of the language. This allows the system to be self-supporting: the environment in which programs are composed, compiled and executed can itself be implemented using the same programming language. Functions that are normally controlled by the operating system can then be integrated with the programming language. These include source code control and versioning, source level debugging, controlling the configuration of applications built from multiple components, documentation, etc. A number of workers are currently addressing the problems of supporting the whole software engineering process within an integrated persistent system [20-24]. Type-safe linguistic reflection is needed to implement such a system.

2.3.4 Hyper-programs

Bringing executable programs into the persistent store allows associations between them to be enforced by direct links. It would be beneficial for the associations between executable programs and source programs to be replaced by direct links also, for the same reason, i.e., they could not then be accidentally corrupted. Then each executable program would contain a direct link to its corresponding source program. As an executable program can also contain direct links to other data items in the persistent store, a source program must be able to denote those data items in order to represent the executable program accurately. This requires the use of hyper-programs as source representations.

Figure 8 shows the relationships in a hyper-programming system. Each executable program contains a direct link to its source hyper-program. Each of the other direct links contained in an executable program is duplicated in its corresponding hyper-program.

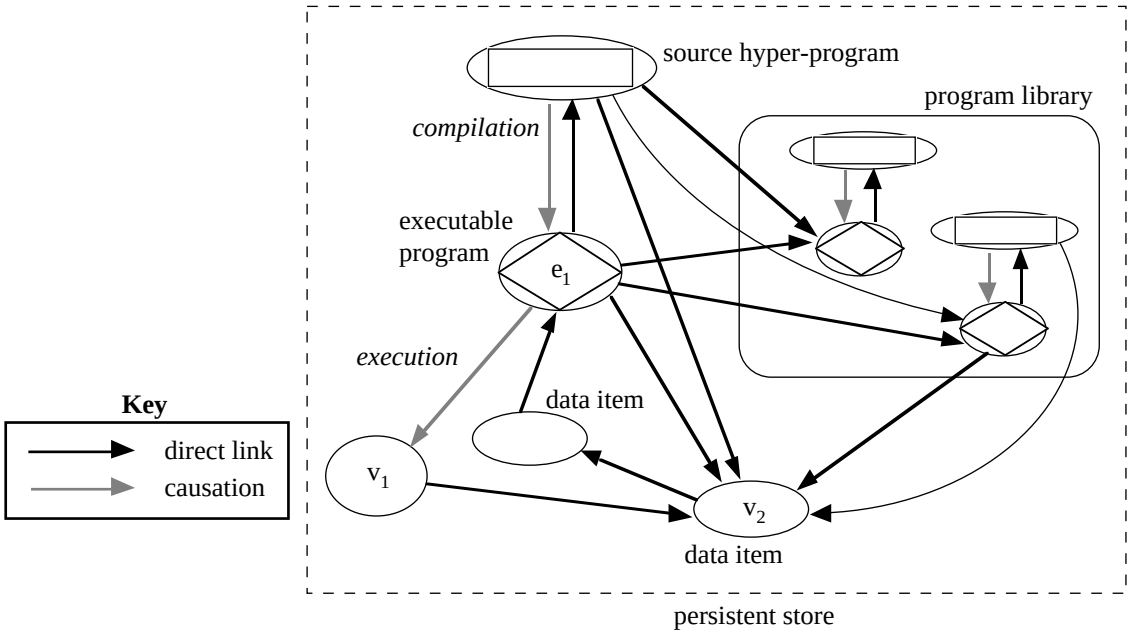

Figure 8: Relationships in a hyper-programming system

To illustrate the necessity of hyper-programs for providing accurate source representations of executable programs, consider the situation where multiple executable programs have direct links to a shared store location as illustrated in Figure 9:

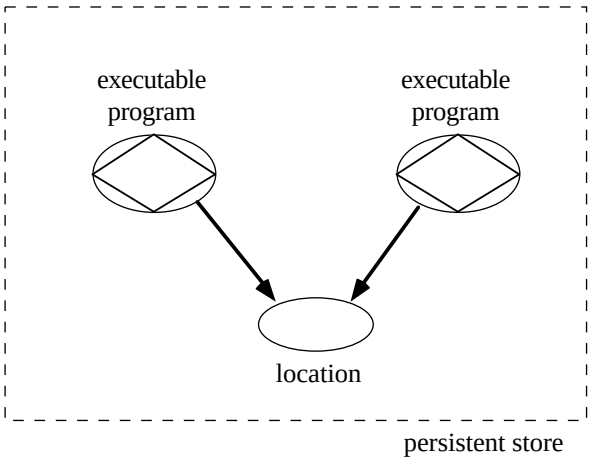

Figure 9: Executable programs sharing a location

The problem in conventional systems arises in supplying separate source programs for each of the executable programs. Unless there is a direct access path to the location from a persistent root, and in general there does not have to be one, conventional source representations do not provide any notation with which the location can be denoted in a source program.

A solution is to change the program notation by introducing hyper-programs as source representations. It is then possible to denote the shared location in the source programs by including tokens for the location. This makes it feasible for every executable program to contain a direct link to its own source hyper-program as illustrated in Figure 10. To preserve the association the source hyper-program is read-only although it can be copied and the copy edited.

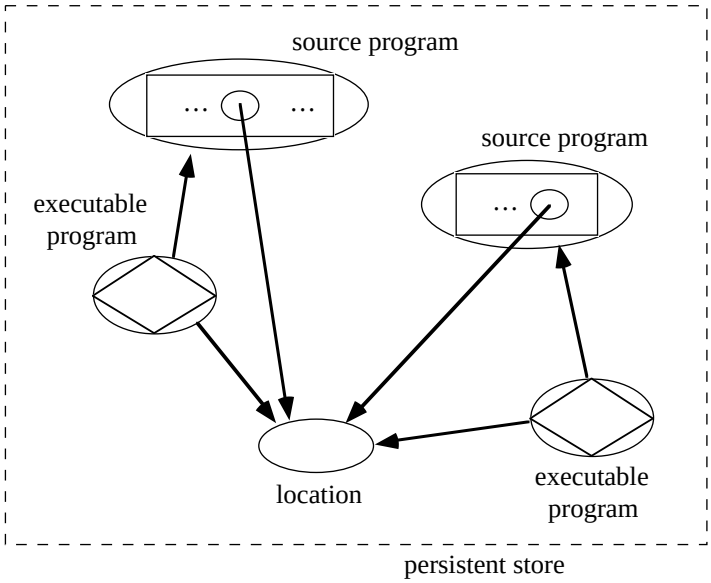

Figure 10: Executable programs with direct links to hyper-programs

Thus the use of hyper-programs as source representations allows associations from executable programs to source programs to be replaced by direct links, improving the robustness of the programming system by eliminating accidental changes to or deletions of source programs.

2.4 Flexible linking mechanisms

Programming languages support a number of different mechanisms for establishing direct links from programs to persistent values, locations and types. The degrees of freedom include constancy or variability, linking to L-values or R-values [25], and the time at which the linking takes place. The focus here is on the range of times available. Some possible times are during program composition, during compilation, during a separate linking phase, and during execution.

The principal varieties of programming system identified earlier were file-based, persistent and hyper-programming systems. Another possibility is a compile-time linking system in which the tokens embedded in a program are associated with data items in the persistent store when the program is compiled rather than when it is written. The linking times possible in each of these systems are shown in Figure 11. From here on it will be assumed that the hyper-programming systems under consideration incorporate facilities for compile-time linking as well as composition-time linking.

System	Linking Time							
	composition		compilation		linking phase		execution	
	program	data	program	data	program	data	program	data
file-based					•			•
persistent					•	•	•	•
compile-time linking			•	•	•	•	•	•
hyper-programming	•	•	•	•	•	•	•	•

Figure 11: Comparison of possible linking times in various systems

File-based systems allow links to existing data to be formed only at run-time. Links to existing programs are formed during a linking phase by copying library programs into the main program. In persistent systems a linking phase can be implemented using first class procedures. Since these executable programs are a form of data, linking to both programs and data can be performed either at link-time or run-time. Compile-time linking systems support these same linking times and also allow linking to programs and data at compilation-time.

A hyper-programming system supports all the linking times described. The programmer can specify various linking times as appropriate for different components of an application. Deciding when components should be linked into a main program involves trade-offs between program safety, flexibility and execution efficiency.

Run-time linking gives flexibility as the data (*data* will now be used to denote both programs and other kinds of data) accessed does not have to be present in the persistent store, file system or database before run-time. Indeed the access path to the data may not be known until run-time. Program safety is low since the data may not be present when the program is run, causing a run-time failure. Execution overheads are also higher, in strongly typed systems, since the type of the data must be checked dynamically. This kind of linking is possible in many systems, for example, C, Pascal, Ada, Smalltalk-80 [26], PS-algol, Napier88.

A distinct linking phase occurs between compilation and execution in some file-based systems, involving the copying of other executable or intermediate programs into the main executable program. A similar effect can also be achieved in persistent languages with higher-order procedures, where all types of data may be linked into an executable program before run-time. This provides improved safety and efficiency over run-time linking, since checks for the data's existence and type are performed before run-time. Flexibility is reduced since its use requires the data to be present earlier.

Linking at compilation-time increases safety and efficiency, bringing checks further forward in time, and reduces flexibility correspondingly. With this mechanism the data linked into an executable program is fixed.

Composition-time linking is the least flexible of the alternatives described as the data linked to must be present at the time that the program is written. It offers the most safety since access to the data is always maintained once it is linked into the source code, even if the source code is edited and re-compiled. This is not true of the other linking styles

where editing of the source code requires all links to be re-established. Efficiency is slightly increased overall since the access path to the data, whether it is expressed by textual code or by user gesture, need be followed only once, at composition-time, and not on every re-compilation.

2.5 Program succinctness

Persistent systems offer significant savings over non-persistent systems regarding the data access code required. One empirical study concluded that 30% of the code in a large set of commercial non-persistent programs was dedicated to transferring data to and from an external storage system [27]. Recent measurements of Napier88 programs have suggested that these access specifications occupy around 13% of program code [28], a considerable reduction on 30%. The intellectual effort required to write the code is also significant: in writing access specifications in a persistent system the programmer is not concerned with programming transformations between structured and flattened formats.

Hyper-programming gives a further improvement in conciseness as the access specifications can in some cases be replaced by tokens that denote persistent data items. The information that was specified in the access specifications is provided through the interactive gesturing by which the programmer points out data items to be linked in. The measurements of Napier88 programs found around 20% of identifiers referring to persistent data. Further work is required to measure the proportion of this data that is available for linking at hyper-program composition time.

Figure 12 summarises the nature of the persistent data access code that appears in source programs in the various cases:

System	Access path code
non-persistent	file access + importing + exporting
persistent	access path + type description
hyper-programming (data present at composition time) (data not present at composition time)	token access path + type description

Figure 12: Comparison of access path code

2.6 Procedure representations

Since hyper-programs can contain direct links to values and locations in the persistent store they can be used to represent executable programs, including those with links to shared locations. This provides a convenient representation format for procedure values.

As described earlier, associations between executable programs and source programs can be replaced by direct links. When a procedure value is created the compilation system can insert a direct link to its source hyper-program. Given referential integrity, the source code will then remain accessible for as long as the procedure value.

The use of hyper-program source representations allows browsing tools to display meaningful representations of procedure values, showing both source code and direct links to persistent data items. This may aid software reuse since documentation in the

form of the original source code can be made available for every procedure value in the persistent store.

Hyper-programs allow separate procedure source representations since shared locations can be denoted by tokens. A further consequence is that one of a group of procedures that share values or locations can be replaced by a refined version without the need to replace the others. This reduces the cost of modifying applications that are composed of multiple procedures.

The use of hyper-programs to represent procedures with shared locations will be illustrated with an example. Figure 13 shows a Napier88 program that places in the persistent store two procedures that share an integer location:

```
let i := 0

in PS() let inc := proc() ; i := i + 1
in PS() let get := proc( → int ) ; i
```

Figure 13: Procedures with a shared location

The program first initialises an integer variable *i* with the value 0. It then creates two persistent procedures that operate on *i*, the first incrementing it by 1 and the second returning its current value. The procedures are made persistent by declaring them in the context of the persistent root environment, obtained by calling the pre-defined procedure *PS*. Although the store location corresponding to the variable *i* is not declared in the persistent environment, it will persist because it is reachable from the procedures *inc* and *get* which are themselves persistent. The result of executing this program is that the persistent store contains the two procedures and the shared integer location which is not directly accessible from the persistent root.

Figure 14 shows the links between the procedures, their hyper-program source representations and the shared location:

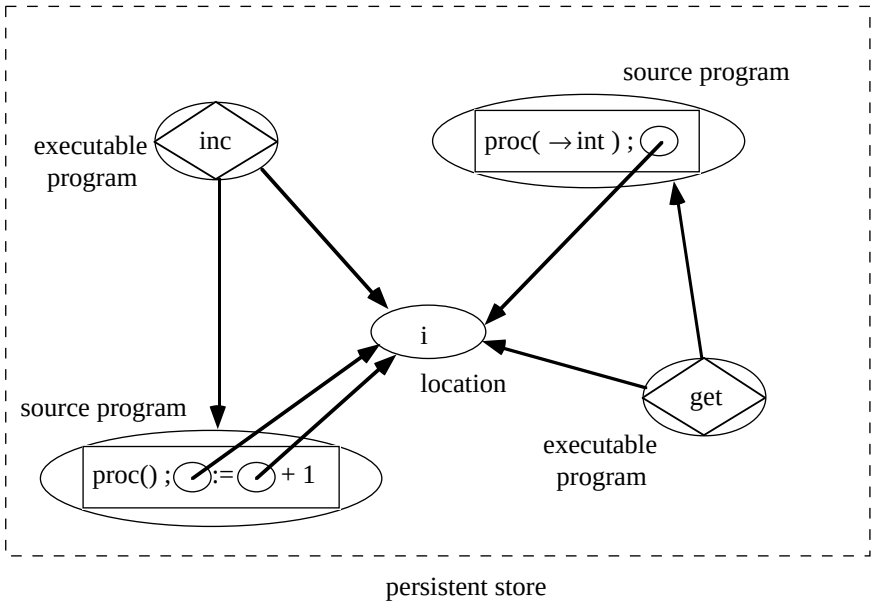

Figure 14: Hyper-programs with a shared location

The use of hyper-program source representations for procedures in this way avoids having to replace all procedures that share locations when a single one is changed. Another advantage is that the same shared locations are retained after the replacement of a procedure. Without hyper-program source representations not only do all the procedures have to be replaced in order to preserve sharing, but new shared locations must be created and the values that were previously shared copied into the new locations.

3 Hyper-worlds

There are a number of components that a persistent programming environment should support if it is to provide for the software engineering process as a whole. These include:

- program composition, compilation and execution;
- storing of source and compiled versions of programs;
- debugging;
- documentation;
- decomposition of large application programs into components, and organisation of those components;
- navigating the persistent store to locate programs and other data with given attributes;
- querying of the types of programs and data in the persistent store.

The model of hyper-programming as described so far allows source programs to contain links to any other data in the persistent store. In large scale systems this generality may lead to several problems. Firstly, the store may become intellectually unmanageable as the number of links increases. Secondly, evolution of application programs by substituting new versions of their components becomes difficult to manage if unrestricted linking to the components is permitted—it may be necessary to locate each data item linked to the component being substituted and determine whether a new version of the data item is required in turn. In addition the model described does not provide a uniform framework for storing meta-data about application components.

One research topic is the provision of additional structure over a basic hyper-programming system to address these needs. The *hyper-world* model offers the programmer a loose coupling mechanism to offset the disadvantages of the tight coupling made possible by hyper-programming. In this model, based in part on that described in [29], the persistent store is partitioned into a number of application spaces or hyper-worlds. Each hyper-world contains the program components and data used by an application, and a schema that describes their relationships. Each hyper-world has a single visible component which may be linked to from outside the hyper-world; no other components inside the hyper-world may be linked to from outside.

The schema includes documentation information, a type description and hyper-program source for each component. It also includes a representation of the component linking topology, and a list of type definitions local to the hyper-world. This allows the programmer to perform various queries over the components, and to determine the implications of replacing a component with a changed version.

The partitioning supported by hyper-worlds may reduce problems such as keeping track of inter-component links to a manageable scale, by restricting the region of interest from the entire persistent store to the hyper-world. It may also allow type-checking to be performed more efficiently.

Figure 15 shows a representation of a persistent store containing nested hyper-worlds and linked components:

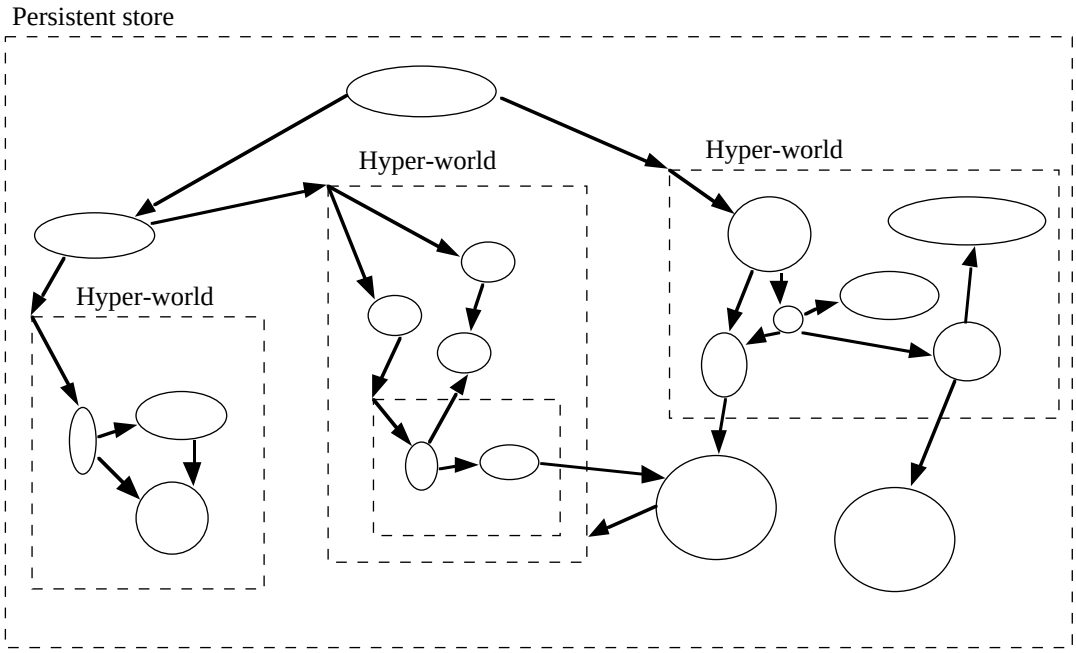

Figure 15: A store with hyper-worlds

4 Implementation status and further research

A prototype Napier88 hyper-programming system has been developed [30], building on earlier systems developed at St Andrews and Adelaide [22, 23].

Although the concept of hyper-programming has been illustrated using Napier88 it is not restricted to that language. It could be implemented for any language that supports first class procedures, orthogonal persistence, run-time linguistic reflection and graphical user interface tools.

There are several avenues for further research in hyper-programming:

Hyper-worlds: the model described above will be further developed and implemented.

Reflective programming: the implementation of the hyper-programming environment is founded on the reflective facilities of Napier88: the ability to write a program (the programming environment) that constructs another program (the code composed in the environment by the user) and compiles it dynamically. This is an implementation issue and one that the user need not be aware of. However, run-time reflection can be a more generally useful programming tool, and some current research addresses the problems of how the programmer can express in a comprehensible way the computations that produce new programs [31]. The possibilities of writing reflective programs that produce hyper-programs will be investigated.

Programming by gesture: in the existing systems values can be browsed by gesturing with the mouse but to achieve any updates to the store the user must write and evaluate new code. Another line of research is to investigate the possibilities of performing more general computation over the store via a ‘direct manipulation’ approach. It would not be hard to provide in an ad-hoc manner ways of performing specific predefined actions; the challenge is to develop some more general model.

5 Conclusions

There are many situations when the programmer writes code to access data items in the persistent store, knowing that those data items are present in the store at the time of writing. This paper has shown how data can be linked directly into a source program as opposed to the program containing instructions on how to link to it at run-time. This gives the benefits provided by interactive languages: greater program safety as there is no danger of losing access to the data during the time between writing and execution, and better efficiency as run-time type and access path checks are factored out. The flexibility of being able to link to the store dynamically when required is retained.

An analysis has been given of the program entities and their inter-relationships in a hyper-programming system, and compared to those found in file-based and existing persistent systems. A number of benefits of using hyper-programs have been described. These include being able to: perform program checking early; enforce associations from executable programs to source programs with direct links; support an increased range of linking times; reduce program verbosity; and provide source representations for procedure closures.

The user interface of a prototype hyper-programming system has been outlined. A store browser is used by the programmer to navigate the persistent store and identify data items. Tokens that denote these data items can then be incorporated into the hyper-program under construction. Facilities for viewing and editing hyper-programs are provided.

Finally a framework, called hyper-worlds, has been proposed for supporting 'programming in the large' in the context of a hyper-programming system. It allows the programmer to impose a degree of partitioning on the persistent store, in order to aid intellectual manageability and improve execution efficiency.

6 Acknowledgements

This work was supported by ESPRIT III Basic Research Action 6309 – FIDE 2 and SERC grant GR/F 02953. Richard Connor is supported by SERC Postdoctoral Fellowship B/91/RFH/9078. Alan Dearle and Alex Farkas are supported by ARC grants "Controlled Evolution" and "Browsing in Persistent Integrated Programming Environments", Adelaide University grant "Persistent Integrated Programming Environment" and by the DSTO. We thank Dave Stemple for his useful comments.

7 References

1. Farkas AM, Dearle A, Kirby GNC, Cutts QI, Morrison R, Connor RCH. Persistent Program Construction through Browsing and User Gesture with some Typing. In: Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy, 1992

2. Stemple D, Stanton RB, Sheard T, Philbrow P, Morrison R, Kirby GNC, Fegaras L, Cooper RL, Connor RCH, Atkinson MP, Alagic S. Type-Safe Linguistic Reflection: A Generator Technology. University of St Andrews Report CS/92/6, 1992

3. PS-algol Reference Manual, 4th edition. Universities of Glasgow and St Andrews Report PPRR-12-88, 1988

4. Morrison R, Brown AL, Connor RCH, Dearle A. The Napier88 Reference Manual. University of St Andrews Report PPRR-77-89, 1989

5. Altmann RA, Hawke AN, Marlin CD. An Integrated Programming Environment Based on Multiple Concurrent Views. Australian Computer Journal 1988; 20,2:65-72

6. Atkinson MP, Lécluse C, Philbrow P, Richard P. Design Issues in a Map Language. In: P. Kanellakis and J. W. Schmidt (ed) Bulk Types & Persistent Data. Morgan Kaufmann, 1991, pp 20-32

7. Wirth N. The Programming Language Pascal. Acta Informatica 1971; 1,35-63

8. Reference Manual for the Ada Programming Language. U.S. Department of Defense Report ANSI/MIL-STD-1815A, 1983

9. Kernighan BW, Ritchie DM. The C programming language. Prentice-Hall, 1978

10. Ritchie DM, Thompson K. The UNIX Time-Sharing System. The Bell System Technical Journal 1978; 63,6:1905-1930

11. Albano A, Cardelli L, Orsini R. Galileo: a Strongly Typed, Interactive Conceptual Language. ACM ToDS 1985; 10,2:230-260

12. Albano A, Ghelli G, Orsini R. The Implementation of Galileo's Values Persistence. In: M. P. Atkinson, O. P. Buneman and R. Morrison (ed) Data Types and Persistence. Springer-Verlag, 1988, pp 253-263

13. Brown AL, Mainetto G, Matthes F, Müller R, McNally DJ. An Open System Architecture for a Persistent Object Store. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 766-776

14. Matthes F, Müller R, Schmidt JW. Object Stores as Servers in Persistent Programming Environments—The P-Quest Experience. ESPRIT BRA Project 3070 FIDE Report, 1992

15. Davie AJT, McNally DJ. Statically Typed Applicative Persistent Language Environment (STAPLE) Reference Manual. University of St Andrews Report CS/90/14, 1990

16. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott WP, Morrison R. An Approach to Persistent Programming. Comp. J. 1983; 26,4:360-365

17. Atkinson MP, Morrison R. Persistent First Class Procedures are Enough. In: M. Joseph and R. Shyamasundar (ed) Lecture Notes in Computer Science 181. Springer-Verlag, 1984, pp 223-240

18. Atkinson MP, Morrison R. Procedures as Persistent Data Objects. ACM ToPLaS 1985; 7,4:539-559

19. Atkinson MP, Morrison R. Integrated Persistent Programming Systems. In: Proc. 19th International Conference on Systems Sciences, Hawaii, 1986, pp 842-854

20. Dearle A, Cutts QI, Connor RCH. An application architecture using type-safe incremental linking. Submitted for publication, 1992

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

22. Farkas AM. ABERDEEN: A Browser allowing intERactive DEclarations and Expressions in Napier88. University of Adelaide Report Honours Project, 1991

23. Kirby GNC, Cutts QI, Connor RCH, Dearle A, Morrison R. Programmers' Guide to the Napier88 Standard Library, Edition 2.1. University of St Andrews, 1992

24. Dearle A, Marlin CD, Dart P. A Hyperlinked Persistent Software Development Environment. In: Proc. Hyper-Oz '92: A Workshop on Hypertext Activities in Australia, Adelaide, Australia, 1992

25. Strachey C. Fundamental Concepts in Programming Languages. Oxford University Press, Oxford, 1967

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

27. IBM Report on the Contents of a Sample of Programs Surveyed. IBM, San Jose, California, 1978

28. Sjøberg D. Measuring Name and Identifier Usage in Napier88 Applications. ESPRIT BRA Project 3070 FIDE Report FIDE/92/37, 1992

29. Wile DS, Allard DG. Worlds: An Organizing Structure for Object-Bases. In: Proc. 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Palo Alto, California, 1986

30. Kirby GNC. Reflection and Hyper-Programming in Persistent Programming Systems. Ph.D. thesis, University of St Andrews, 1992

31. Kirby GNC. Persistent Programming with Strongly Typed Linguistic Reflection. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 820-831