Current Directions in Hyper-Programming

R. Morrison[†], R.C.H. Connor[¶], Q.I. Cutts[¶], A. Dearle[*],
A. Farkas[+], G.N.C. Kirby[†], R. McGettrick[*] & E. Zirintsis[†]

†School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, Scotland
{ron, graham, vangelis}@dcs.st-and.ac.uk

¶Department of Computer Science, University of Glasgow,
Glasgow G12 8QQ, Scotland
{richard, quintin}@dcs.gla.ac.uk

*Department of Computing Science and Mathematics,
University of Stirling, Stirling, FK9 4LA, Scotland
{al, rmc}@cs.stir.ac.uk

+Vision Systems Ltd,
Adelaide, S.A., Australia
Alex.Farkas@vsl.com.au

Abstract. The traditional representation of a program is as a linear sequence of text. At some stage in the execution sequence the source text is checked for type correctness and its translated form is linked to values in the environment. When this is performed early in the execution process, confidence in the correctness of the program is raised. During program execution, tools such as debuggers are used to inspect the running state of programs. Relating this state to the linear text is often problematical. We have developed a technique, *hyper-programming*, that allows the representations of source programs to include direct links (hyper-links) to values, including code, that already exist in the environment. Hyper-programming achieves our two objectives of being able to link earlier than before, at program composition time, and to represent sharing and thus closure and through this the run-time state of a program. This paper reviews our work on hyper-programming and proposes some current research areas.

1 Introduction

Fig. 1, taken from [1], shows an example of a Napier88 hyper-program. The program source, which is itself a persistent object, comprises text and hyper-links to other objects in the persistent store.

The first hyper-link is to a persistent first-class procedure value *writeString* which writes a prompt to the user. The program then calls another procedure *readString* to read in a name, and then finds an address corresponding to that name. This is done by calling a procedure *lookup* to find the address in a table data structure linked into the hyper-program. The address is then written out. Note that the code objects (*readString*, *writeString* and *lookup*) are denoted using exactly the same mechanism as data objects (the table)[1] and all of these are external to the hyper-program but within the persistent environment.

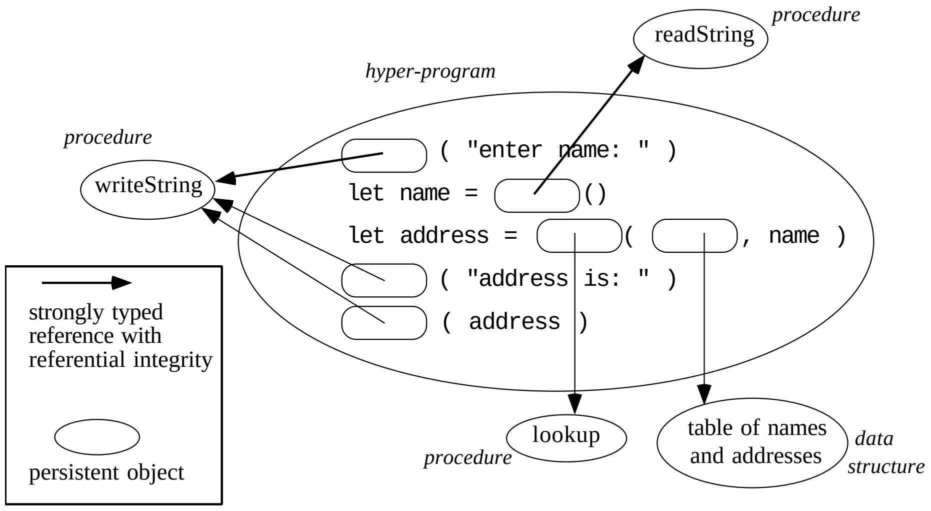


Fig. 1. A Napier88 Hyper-Program

A requirement for hyper-programming is the presence of an external value space to which bindings can be constructed during program composition. The external source may be provided by a persistent store, a file system or any other mechanism such as the WWW. No matter which external source is used, a fundamental change in the nature of the source program has taken place since it now contains both text and hyper-links to values in the environment. This non-flat representation of the program source challenges our traditional notions of what constitutes a computer program. The reason for the name *hyper-program* is the analogy with hyper-text which is also non-flat and contains both text and hyper-links to other hyper-text.

The major issue in building hyper-programming systems concerns the semantics of the hyper-links, such as:

- what can a hyper-link refer to?
- what guarantees can be made about a hyper-link's referent data?
- how are hyper-links typed and when does type-checking occur?

The degrees of freedom regarding what a hyper-link can refer to depend upon the programming language semantics and the measure of openness is the system. Normally hyper-links will be able to refer to all language first class values. Second class entities, not in the value space such as types, may also be conveniently hyper-linked depending on the flavour of the language. Update may be accommodated through

[1] Note also that the names used in this description of the hyper-links have been associated with the objects for clarity only, and are not part of the semantics of the hyper-program.

hyper-links by linking to locations, which may or may not be first class values. More interesting is the extent to which hyper-links may refer to values created independently of the system, such as Web pages and DCOM objects. Furthermore the openness of the system can be extended by making the hyper-program representation open for other tools to manipulate.

Referential integrity in a hyper-programming system means that once a hyper-link is established it is guaranteed by the system to exist and to be the same value when the hyper-link is executed. While this guarantee may be provided by a strongly typed persistent object store, it may also be expensive to provide in a distributed system. Variations therefore include the hyper-link being valid but not necessarily referring to the original value, and the hyper-link referring to a copy of the original. This may only be a problem where object identity is important such as in sharing semantics. A hyper-program may therefore display a range of failure modes from not failing to failure from the hyper-link being no longer valid.

The final issue is how hyper-links are typed, if at all. We will assume that for the present that they are. The interesting aspect of type checking is that the contract between the program and the referenced value may now take on a different agreement procedure. Instead of the program asserting the type of the hyper-link and the type checking system ensuring that the hyper-link has the correct type when it is used, the reverse may be used. That is the hyper-link knows its own type and therefore when it is used the program can be made to conform to this type. Statically this removes the need for type specifications for hyper-links in hyper-programs and dynamically it means that the program may be in error rather than the hyper-link.

This paper reviews our work on hyper-programming, discusses the advantages of the technique and proposes some current research areas. These include presenting a single representation of data and code throughout the software process; adapting hyper-programming to persistent contexts that do not enforce referential integrity, such as the WWW; and implementing and using hyper-programming in standardised languages and inter-operability mechanisms.

2 Motivations & Previous Work

Our work on hyper-programming is motivated by a belief that programming language systems could provide better support for the software engineering process than they do at present. In particular, consider the traditional *compose-compile-link-execute* cycle of program development as illustrated in Fig. 2.

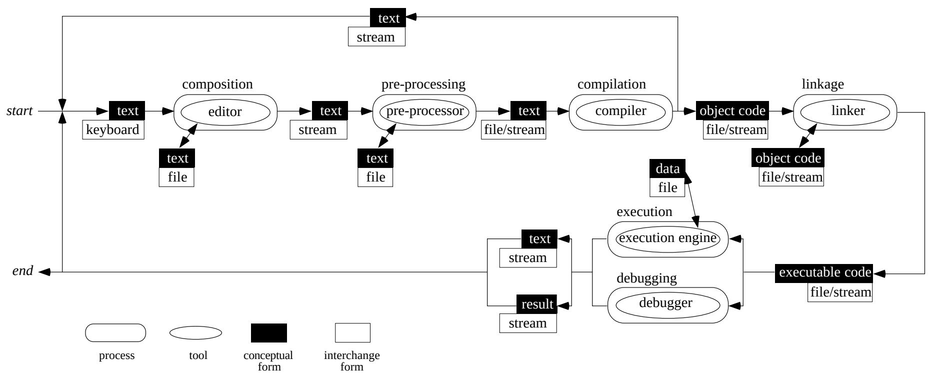


Fig. 2. The Traditional Compose-Compile-Link-Execute Cycle

In precis, a program is composed using a text-editor; compiled using a compiler, which may also link in other source text; linked with other pre-compiled code; and finally executed where it may link to persistent data such as files. During execution, other tools such as symbolic debuggers and run-time browsers may be used to inspect the running state of the program. Thus there are four main processes: composition, compilation, linking and execution each with their appropriate tools such as text-editors, compilers, linkers, debuggers and browsers. Each tool operates on a particular translated version of the program such as source text, object code or executable code.

There are two obvious questions that may be asked about the compose-compile-link-execute cycle. They are:

- why are there so many processes and translated forms of the program? and
- what level of detail should the user see?

For the systems programmer the processes and translated forms provide the necessary level of control over the cycle. The translated forms allow common tools, such as optimisers, to be used even where the original forms are from disparate sources. The processes are necessary for manipulating the translated forms.

From the applications programmer's point of view, the processes and translated forms often constitute noise in the execution cycle and a distraction from the task of constructing the system. Modern programming environments, such as CodeWarrior [2], attempt to hide this level of detail from the applications programmer. Hyper-programming is a further step in this direction and the paper explores how effective the concept can be in different environments.

2.1 Constructing Hyper-programs

The primary motivation for hyper-programming is to allow the user to compose programs interactively [3, 4], navigating the environment and selecting data items, including code, to be incorporated into the programs. This removes the need to write access specifications for extant data items that are used by a program. For example, in a file system it may be a path name, and in a persistent object store it may be a path to an object from a root of persistence.

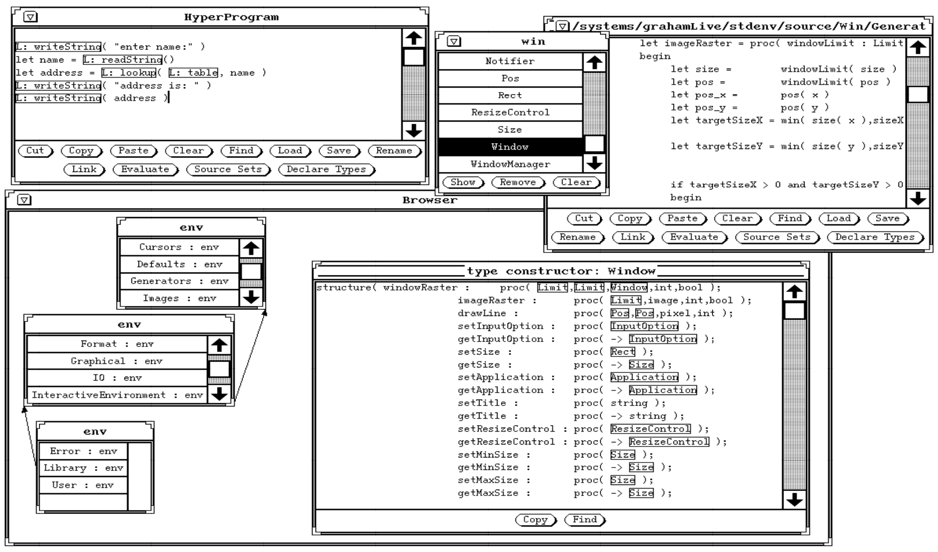
Our first attempts at constructing a hyper-programming system were conducted in the Napier88 persistent programming environment. The strongly typed persistent object store guaranteed referential integrity of the hyper-links. Existing languages that allow a program to link to persistent data items, including files, at any time during its execution 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 position. When a program is compiled the compiler checks that subsequent use of the data is compatible with this 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 and the programming system could obtain all the information in the access specification by inspecting the data item at that time.

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. There are two advantages to this early composition-time linking. Firstly, errors that may occur in programs due to the access specification being invalid at the time of execution are completely avoided. This may occur where the store topology has changed and the access path no longer exists, even if the object does; where the object has been deleted; or where the object has been replaced by one of a different type. In all cases the contract between the program and the persistent store has been broken and the program may not execute safely.

In the hyper-programming system the hyper-link is direct to the object and is guaranteed to be valid, at the time of the program execution, by the persistent store's referential integrity. Thus if the topology of the store changes, the link will still be valid; the object may not be deleted since the hyper-program still has access to it; and it may not change its type.

Fig. 3 shows an example of the user interface that might be presented to the user by a hyper-program editing/browsing tool. The editor window (top-left) contains embedded buttons representing hyper-program links; when a button is pressed the corresponding object is displayed in a browser window (lower region).



```
L: writeString( "enter name:" )
let name = L: readString()
let address = L: lookup( L: table, name )
L: writeString( "address is: " )
L: writeString( address )
```

```
    let imageRaster = proc( windowLimit : Limit
    begin
        let size =         windowLimit( size )
        let pos =          windowLimit( pos )
        let pos_x =        pos( x )
        let pos_y =        pos( y )
        let targetSizeX = min( size( x ),sizeX

        let targetSizeY = min( size( y ),sizeY


        if targetSizeX > 0 and targetSizeY > 0
        begin
```

```
structure( windowRaster :     proc( Limit,Limit,Window,int,bool );
                   imageRaster :      proc( Limit,image,int,bool );
                   drawLine :         proc( Pos,Pos,pixel,int );
                   setInputOption :   proc( InputOption );
                   getInputOption :   proc( -> InputOption );
                   setSize :          proc( Rect );
                   getSize :          proc( -> Size );
                   setApplication :   proc( Application );
                   getApplication :   proc( -> Application );
                   setTitle :         proc( string );
                   getTitle :         proc( -> string );
                   setResizeControl : proc( ResizeControl );
                   getResizeControl : proc( -> ResizeControl );
                   setMinSize :       proc( Size );
                   getMinSize :       proc( -> Size );
                   setMaxSize :       proc( Size );
                   getMaxSize :       proc( -> Size );
```

Fig. 3. User Interface to a Hyper-Program Editor

The hyper-links to persistent values are placed in the hyper-program by selecting each value with the store browsing tool and then pressing the *Link* button. In Napier88, 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 the link at the current text position, represented by a light-button.

2.2 Safety and Efficiency

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.

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.

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 link 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. The hyper-program will be unaffected if the access path is then removed.

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 link is compatible with the use of the link 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 links 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 link 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 links in a hyper-program can vary but their types will always remain compatible. Where a link 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.

2.3 Experience

The benefits of hyper-programming described in [1, 3, 4] may be summarised as:

- being able to perform program checking early
- support for source representations of all object closures
- being able to enforce associations from executable programs to source programs
- availability of an increased range of linking times

- increased program succinctness
- increased ease of program composition

3 Current Work

3.1 Options for Further Development

Hyper-programming as described in the previous section is implemented in Napier88 [5] and using a persistent form of Java, PJama [6]. Both implementations are based on the use of a closed-world, single-language, programming environment. The principle advantage of this is the degree of control that can be exercised over the data and code within the environment. In particular, a type system can be enforced over the entire lifetime of the data and code, and referential integrity can be guaranteed by the environment implementation. Thus, once established, a reference between two components will never become accidentally invalid.

The use of such an environment offers various benefits, as discussed previously, at the cost of limiting flexibility. There are thus two main avenues for further development of the hyper-programming concept:

- to further pursue the benefits of using a closed-world system, accepting the limitations that this implies; and
- to investigate how far the closed-world restrictions may be relaxed to increase flexibility, while retaining at least some of the original benefits of hyper-programming.

Sections 3.2 to 3.4 describe three areas of research based on a closed-world platform: *hyper-code*, in which a single uniform representation of code and data is presented throughout the programming life-cycle; support for application evolution based on tracking relationships between system components using referential integrity; and statically checkable dependant types. Some other areas in which a closed-world could be exploited, although not discussed further here, include:

- version control, configuration management and documentation systems [1]; and
- debugging, profiling and optimisation [7].

Sections 3.5 and 3.6 examine two ways in which the hyper-program platform constraints may be usefully relaxed: constructing programs over an unreliable network such as the World Wide Web; and hyper-programming using commercially significant languages and inter-operability standards, such as C++ [8], CORBA [9], DCOM [10] etc.

3.2 Hyper-Code

One of the original motivations for persistent programming was to remove the conceptually unnecessary distinction between short-term and long-term data [11]. This was followed by the recognition that code and data can usefully be treated in a uniform way [12]. Hyper-programming itself involved a further unifying step in which source programs themselves became persistent data, along with the compilers, editors and other tools with which they were manipulated [4]. There has thus been a progression of attempts to encompass ever more of the disparate entities that comprise a Persistent Application System (PAS) within a unified framework.

Visual interaction with persistent data, such as that provided by generic object browsing systems [13-19], 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 persistent data such as held in 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.

Current work on *hyper-code* aims to complete the progressive integration of PAS entities [20], by presenting the programmer with a single representation form for all code and data throughout all stages of the programming process. These stages include at least object store browsing, program construction, execution, debugging and maintenance. The single representation form is based on source code, the argument being that all other forms of code and data are used for pragmatic implementation-driven reasons, rather than being conceptually necessary. Since the representation must be able to accommodate closures, by necessity it is a hyper-program form that can include direct links.

Hyper-code provides the basis for a new style of editor that includes three unifying concepts, the combination of which makes the editor the only mechanism that is required for interaction with the database system. The three important unifying concepts are:

- 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.
- Source code is treated not as a fundamental building block within the programming system, but instead as a transient text-based view of a value. The source does not have a conceptual permanent existence within the system, but is apparently generated from any value that may be browsed.

- 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.

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, 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. In constructing a program, the programmer writes hyper-code. During execution, during debugging, when a run time error occurs or when browsing existing programs, the programmer is presented with, and only sees, the hyper-code representation. Thus the programmer need never know about those entities that the system may support for reasons of efficiency, such as object code, executable code, compilers and linkers. These are maintained and used by the underlying system but are merely artifacts of how the program is stored and executed, and as such are completely hidden from the programmer.

A consequence of the above is that the hyper-code 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 programmer may thus concentrate on the inherent complexity of the application rather than on that of the support system.

Hyper-Code Operations. The previous hyper-programming implementations in Napier88 [21] and Java [19] approach this ideal, but fall short in two ways. Firstly, the programmer is aware of a distinction between the source and compiled versions of code entities; and secondly, code and data entities are manipulated differently, using an editor and an object browser respectively. Hyper-code removes these distinctions. In the first case, the occurrence of system activities such as compilation and linking is hidden, since they are implementation details—the view presented to the programmer is one of source level interpretation. In the second case, all interaction with the hyper-code system is via a single hyper-code editor that fulfils the functions of both the browser and editor in the previous systems. The hyper-code editor supports only the following operations:

- *evaluate*: this executes a selected fragment of hyper-code and returns the result, if any, as a new hyper-code fragment;
- *explode*: this expands a selected link in a hyper-code fragment to show more detail, which is itself expressed in the form of hyper-code;
- *unexplode*: this contracts an exploded link back to its original form;
- *edit*: this includes all conventional editing facilities;
- *get root*: this returns a selected persistent root, as a hyper-code fragment.

When composed, these operations are sufficient to support all program construction, execution and persistent object browsing activities. Note that various system activities are implicit in the operations. For example, the implementation of the *evaluate* operation involves syntax checking, compilation and invocation of the selected code representation.

The semantics of the hyper-code operations can be defined in terms of four abstract operations, which are *reflect*, *reify*, *execute* and *transform*. As shown in Fig. 4, these operate on two distinct domains: the domain of persistent hyper-code entities and the domain of hyper-code representations. The former domain contains all of the first class values defined by the programming language, together with various non-first-class entities for which it may be useful to have representations, such as types, classes and executable code. Only the latter domain, that of hyper-code representations, is made explicit to the programmer.

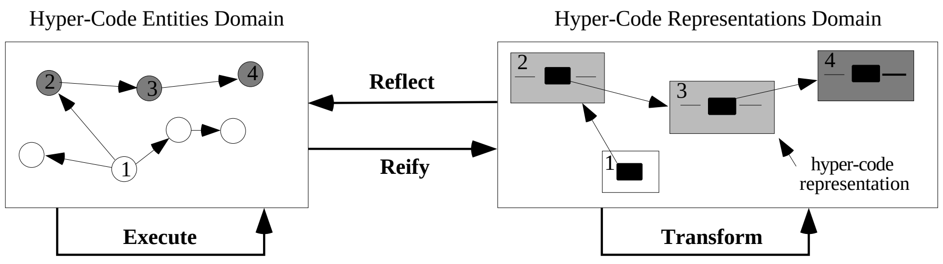

Hyper-Code Entities Domain
Hyper-Code Representations Domain
Reflect
Reify
Execute
Transform
hyper-code representation
1
2
3
4

Fig. 4. Hyper-code Domains and Abstract Operations

The *reflect* and *reify* abstract operations simply map between the hyper-code entities and their representations. The *execute* operation takes place within the hyper-code entities domain: it involves the execution of an executable entity, potentially with side-effects on the domain. Correspondingly, the *transform* operation takes place within the representation domain, involving the manipulation of hyper-code representations. The hyper-code operations can be understood in terms of the abstract operations as follows:

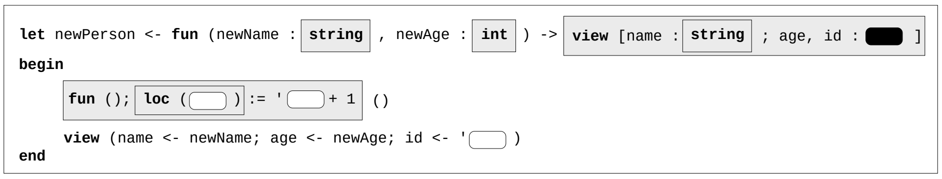
- *evaluate* first *reflects* a hyper-code representation to a corresponding hyper-code entity. If that entity is executable it is *executed*. If the execution produces a result entity, or if the original entity is non-executable, that entity is *reified* to produce a result representation.
- *explode* and *unexplode* both *reflect* a hyper-code representation to a corresponding hyper-code entity, and then *reify* that entity to produce a more or less detailed result representation, respectively.
- *edit* involves *transformation* of an existing or null hyper-code representation into a new representation.
- *get root* involves *reification* of a hyper-code entity to produce a representation.

It should be stressed that the abstract operations are purely definitional: only the hyper-code representations domain and the hyper-code operations are visible to the programmer.

Hyper-Code Representations. The operations and domains described in the previous section may be applied to an implementation of hyper-code in any suitable language. The precise form of the hyper-code representation (HCR) will vary depending on the syntax of the chosen language, but will be guided by the following criteria that will apply for all languages:

- The HCR must accommodate new programs written in the normal way. This implies that the representation must include pure text as a special case.
- The HCR must support hyper-program links, for the reasons already discussed.
- The HCR must support detailed views of linked entities, to arbitrary levels of detail, in order that the hyper-code editor may subsume the functions of an object browser.
- Since there must only be a single HCR, the detailed views of entities must themselves comprise text and hyper-program links in the same form as could be constructed by the programmer.
- Furthermore, the detailed views should be self-contained and syntactically valid. Thus, for any detailed view of an entity, it should be possible to copy its representation, paste this into a new window, and evaluate it without error. The result of this evaluation will depend on the semantics of the language.

Currently we have designed HCR forms for PJama and ProcessBase[2], and have implemented a prototype in PJama. Fig. 5 shows an example in ProcessBase, in which unexploded links to values are denoted by rounded white rectangles, and unexploded links to types by rounded black rectangles. Exploded links are denoted by shaded rectangles, with the internal details depending on the particular entity. The example shows the definition of a procedure *newPerson*, which takes a name and an age as parameters, and returns a view (record) containing them and a unique id number. The id is obtained by calling another procedure to increment a shared location, and then dereferencing that location.



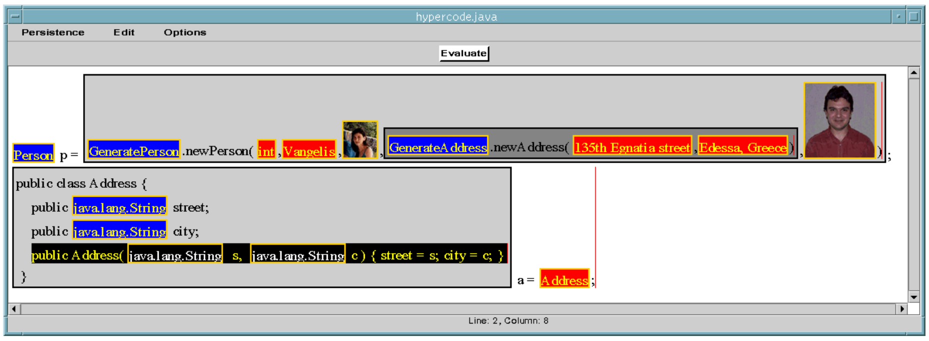
```
let newPerson <- fun (newName : string , newAge : int ) -> view [name : string ; age, id :  ]
begin
    fun (); loc (  ) := '  + 1  ()

    view (name <- newName; age <- newAge; id <- '  )
end
```

Fig. 5. Example of Hyper-Code Representation in ProcessBase

Our HCR design for PJama, an example of which shown in Fig. 6, is similar to that in Fig. 5, although it is less elegant due to the higher number of non-first-class entities to which it must support linking, and the presence of non-public object fields.

[2] A simple persistent language being developed as part of the Compliant Systems Architecture project [22, 23].



```
Person p = GeneratePerson.newPerson( int , Vangelis , , GenerateAddress.newAddress( 135th Egnatia street , Edessa, Greece ) , );
public class Address {
    public java.lang.String street;
    public java.lang.String city;
    public Address( java.lang.String s, java.lang.String c ) { street = s; city = c; }
}
                                                   a = Address;
```

Fig. 6. Example of Hyper-Code Representation in PJama

3.3 System Evolution

Hyper-programming is also the basis for providing new solutions to the problem of schema editing which requires location and translation of affected queries and data [24]. The essential elements are at hand in the hyper-programming system. The schema may keep a record of which programs (queries) and data are associated with particular parts of the schema via secure links. The programs always have hyper-program source and therefore source code and data translation is possible.

The schema evolution mechanism transforms the programs and data affected by a schema edit. This is achieved as follows:

- Locate, from the schema, all affected programs and data.
- For each program which may be affected, obtain its hyper-program.
- Locate the points in the hyper-program which access the changed part of the schema and edit the hyper-program to reflect the new logical schema structure. This will involve establishing new links both to and from the changed part of the schema.
- Update the old program with the new one.
- Update the affected data with new versions.

The extent to which this process can be automated depends upon the complexity of the schema change incurred. The essential point is that all interrogation and manipulation of schema, program and data occurs within a single integrated environment, and may therefore be represented as a meta-level program within that environment.

The mechanism relies heavily upon the self-contained nature of the persistent environment. As all the data and code is held in the same environment as the schema, it is possible to keep not only links from the schema to the data it describes but also reverse links from the schema to programs which bind to particular points of it. The hyper-programming concept makes it possible to map between executable and source representations. The fact that these representations are themselves values within the persistent environment, along with the provision of a compiler in the same environment, makes this strategy possible.

3.4 Dependent Types

In addition to data access checking as described in Section 2.2, 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 [25].

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* [26], instances of which are associations between sets of values. The type of a particular map is dependent on the identity of the procedure that 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.

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.

3.5 Internet Programming

The potential association between the concept of hyper-programming, and the Web, is obvious. The source format of hyper-programs is similar to hyper-text, and the Web provides a well-known hyper-text system over the global autonomous network. The clear appeal, therefore, is to somehow extend the paradigm to make it work in this context.

This appeal, however, is fraught with serious technical difficulty, and it would be over-ambitious and pre-emptive to attempt to document it fully in this paper. We therefore restrict the discussion to an elaboration of the problems involved, and outline strategies which we believe may eventually provide solutions.

Problems exist in the following categories:

- how can program source be represented?
- how can typed data be integrated with the http protocol?
- how could data deriving from other web sources be integrated in a typed computation?
- how can the potential failure of references be made tolerable?

These topics are currently under investigation within the framework of the Hippo project at the University of Glasgow[3]. Here we describe only the direction taken for further investigation within each category.

Program Source. To be properly compatible with the Web, it is necessary to represent hyper-programs in a standard text-based form. In the hyper-programming prototypes that have been built, program source is represented in a proprietary format, manipulated only by specially written editor/browser software. This allows the presentation of the program source to the programmer to be strongly associated with the programming language definition. However, to move to a standard internet treatment, the program source format must be open, textual, and ideally should be HTML itself.

One of the known (and as yet largely unaddressed) problems associated with hyper-programming is how standard language treatments, such as the definition of typing and semantics, can be adapted to the hypertext domain. Widely used methodologies for formal definitions and proofs invariably rely upon a textual source representation; while we can claim properties for hyper-programs on a purely intuitive level, it is not clear how to proceed with elementary proofs within a derived system, to demonstrate beyond doubt that there is no flaw in the soundness of the derived language.

Our proposed solution to these problems is to use a two-level language representation and definition. At a high level, humans can interact with a hypertext source, whereas at a lower level the program is actually represented in HTML, including a standard use of hypertext anchors to represent hyperlinks within programs. This allows standard HTML tools, such as high-level composition tools and browsers, to be used as a human-readable interface over the low-level representation. The low-level representation, using standard HTML, allows text-based protocols to be used to interpret and transport the HTML.

The difficulty with such an approach is how to define the overall system in a manner which gives a clear and formal definition of its semantics. The overall system will be relatively complex, in comparison with existing hyper-programming systems where an intuitive semantics is relatively acceptable, given that the low-level representation is not patent.

One approach to this problem is based on the definition of the two-level programming algebra using linguistic reflection as a language definition technique. This approach is based upon the use of compile-time reflection, as defined in [27]. A subset of HTML may be defined as the core programming algebra, making it possible to define the semantics of both standard language features and hyper-links. A hyper-text view of programs, as may be presented by both specialist program editors and standard browsers, can be defined (using the terminology of [28]) as a reflective sub-language, which is used to generate the HTML-based textual form during static analysis by the programming system implementation.

Using linguistic reflection as a definitional mechanism gives a well-defined formal framework in which hyper-programs can be described using relatively conventional definition techniques. Furthermore, it gives a framework wherein the core definition

[3] www.hippo.org.uk

of hyper-programs is text-based, thus allowing their transportation around the various text-based protocols of the Internet without resorting to ad-hoc translation techniques.

Typed Data. Given a persistent programming language which can be used to program over embedded URLs, the next step is to consider how a URL can be used to refer to typed data, even supposing that the URL refers to data generated by the same programming system. The problem in turn decomposes into three further issues, These are:

- unifying the global persistent namespace with those namespaces used in the Web;
- unifying the representation of the typed persistent data with that commonly used on the Web, namely HTML;
- introducing type system mechanisms which allow the integration of remote, unreliable, and autonomous data with an otherwise static type system.

Each of these presents significant technical challenges, and is not further expanded in this context. Interested readers are referred to [29] for a more detailed exposition of the approach taken; once again, solutions to these problems are still beyond our grasp.

Importing Data. The full potential of a web-based hyper-programming system would only be met if it were possible to include links to data which had been generated by some system other than the particular programming language in use. Once again, this is an enormous issue and can not be addressed in this short space. There are two simple solutions: the first is to read the data as text or MIME, and restrict the typing of such links according to its transmitted classification. This results in a type safe language, assuming the consistent use of the protocol, but does not really address the spirit of the problem. The other simple solution is to publish the format used for the system's own typed data, and ensure it is possible to generate that externally. However any serious uptake of this system then requires the retrospective adoption of a new data standard, which is unlikely to succeed.

The more ambitious goal is to attempt to analyse arbitrary data resulting from an *http* request for appropriate structural content and, if it is suitable, integrate it into a typed computation. The outline of our approach is for the programmer to specify a required type for the binding during the composition process. The URL is duly fetched, and translated into a semi-structured format according to a number of ad-hoc rules.[4] Having achieved a semi-structured representation of the data, the programmer's asserted type is used to derive a subset of the data which corresponds to the same structure. This data is extracted and incorporated into the ongoing computation. An estimation of how well the data fits the expected type is also generated, and may be either returned to the user of the program or used within the running program.

Although we have evidence that the outline given above is possible to engineer [30, 31], and furthermore gives a viable and understandable programming system, each of the steps described presents its own major problems and the production of such an integrated programming system is still beyond current understanding.

[4] The ad-hoc nature of this part of the process can be entirely circumvented when the document is XML, which we perceive to be a rapidly emerging standard for Web information.

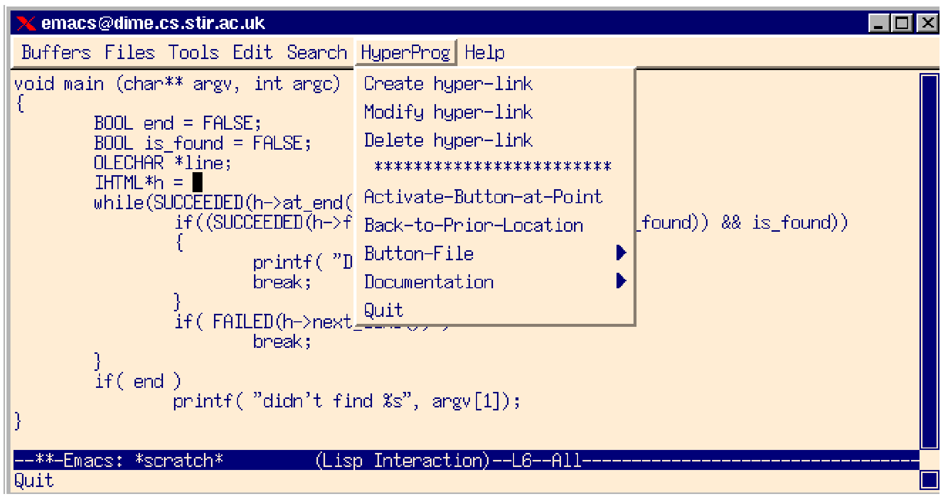
Internet Hyper-Programming? In summary, there is a clear and easy intuition that an extension of the hyper-programming paradigm to encompass the global hyper-text concepts of the World-Wide Web will result in a powerful distributed programming paradigm. While we believe that this is the case, on deeper inspection the technical issues underlying such a paradigm shift are profound. A great deal of work remains to be done before we can be convincing that the extended concept is feasible, whilst retaining a sound and disciplined programming system.

3.6 An Open C++/DCOM Hyper-Programming Environment

In this section we report on an attempt to apply the hyper-programming model in the context of an open system. We chose a DCOM/C++ system for the experimentation for a number of reasons. Firstly, both C++ [8] and DCOM [10] are being used by a large number of programmers to build systems in the real world. Secondly, having programmed with DCOM and C++, we felt there was a high degree of accidental complexity associated with this style of programming that was not intrinsic in the problem domain. We hoped that hyper-programming might be used to simplify the construction of DCOM programs. Finally we were influenced by the HIPPO work of Connor [29] and sought to discover if C++/DCOM programs could be written which had the same flavour as Hippo programs. If this was possible, the power of the many C++ libraries and environments could be used cheaply construct Web utilities. In addition to creating a hyper-programming environment for a commercial system, a deliberate attempt was made to maximise the use of freely available software and to avoid writing new software whenever possible.

Hyper-Program Construction. A DCOM/C++ hyper-program is constructed using two tools: a text editor and a binder. These are used to specify the hyper-program text and the hyper links respectively. The output from these tools is fed into a pre-processor which unifies the source and the links into standard C++ prior to presentation to the gnu-C++ compiler. The pre-processor also creates files and directories for cache maintenance and in some circumstances pre-fetches Web pages.

Editing Environment. The first tool requirement was for a text editor capable of incorporating hyperlinks and suitable for editing programs. Web editing tools such as Netscape Composer and FrontPage do not support the editing of programs since they are intended as HTML composition tools. Consequently Emacs [32] was used with a (then) freely available extension called Hyperbole [33]. Hyperbole supports the inclusion of hyperlinks into documents. In particular, these links can refer to Uniform Resource Locators (URLs), i.e. web pages, and can be clicked on with the mouse. A Hyperbole user works with buttons embedded within textual documents. These buttons may be created, modified, moved or deleted. Each button performs a specific action, such as linking to a file or executing a shell command. Fig. 7 shows a C++ hyper-program being edited with the Emacs/Hyperbole environment.



```
void main (char** argv, int argc)
{
        BOOL end = FALSE;
        BOOL is_found = FALSE;
        OLECHAR *line;
        IHTML*h = 
        while(SUCCEEDED(h->at_end(
                if((SUCCEEDED(h->f               found)) && is_found))
                {
                        printf( "D
                        break;
                }
                if( FAILED(h->next_
                        break;
        }
        if( end )
                printf( "didn't find %s", argv[1]);
}
--**-Emacs: *scratch*           (Lisp Interaction)--L6--All-----------------------------------
Quit
```

Fig. 7. Emacs and Hyperbole

The Hyper-Program Source Code. The program shown in Fig. 8 contains a C++/DCOM hyper-program that finds the telephone number of a member of the Computer Science Department at Glasgow University. It does this by scanning an HTML page denoted by the hyperlink *telephonedirectory*. The program creates a binding denoted by *h* of type *IHTML** to this Web page. The IHTML class shown in Fig. 9 supports a number of operations including the *find_in_line* method which searches lines of the page looking for the sub-string specified in the parameter. If a match is found the line is returned. It also contains a predicate *at_end* indicating that the end of the page has been reached.

```
void main (char** argv, int argc)
{
   BOOL end = FALSE;
   BOOL is_found = FALSE;
   OLECHAR *line;
   IHTML*h = <(telephonedirectory)>;
   while(SUCCEEDED(h->at_end(&end)) && ! end) {
      if((SUCCEEDED(h->find_in_line(
                              argv[1,&line,&is_found)) &&
         is_found)) {
         printf( “Details are %s \n\r”, line ); break; }
      if( FAILED(h->next_line()) ) break;
   }
   if( end ) printf( “didn’t find %s", argv[1]);
}
```

Fig. 8. A C++/DCOM Hyper-Program

```
interface IHTML : Iunknown
{
   HRESULT display_line();
   HRESULT openURL([in, string] char* filename);
   HRESULT next_line();
   HRESULT find_in_line([in, string] char* name,
       [string, out] OLECHAR** line,[out] int* isfound);
   HRESULT at_end([out] int *i);
}
```

Fig. 9. MIDL Definition of the IHTML Interface

The code shown in Fig. 8 is standard DCOM/C++ except for the line,

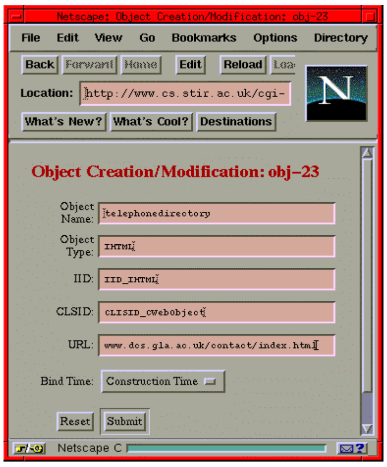
```
IHTML*h = <(telephonedirectory)>;
```

which has to be replaced with standard C++, as described above this task is performed by the pre-processor. The code sequence into which this hyper-link is expanded depends on the binding style specified in the binder. This is described the next section.

Creating Bindings. Using the Hyperbole environment, bindings can be made to any Web based data. However, this does not address the need to specify attributes associated with those links such as programming language type, external data type, the location of the data being bound and binding time. To allow hyper-programmers to specify and view bindings, a Web interface to a *binder* has been created and is shown in Fig. 10.

The binder permits users to specify a name for a hyper-link. This is used to match the hyper-links entered in the editor with bindings specified in the binder. The next field is the type of the object in the programming language context. In the current implementation this field contains a string which is used to specify the programming language type of the target object. This field is strictly unnecessary since it could be automatically generated but makes the generated code more readable. The next field, *IID*, is used to specify the type (interface) of the object being linked to. In the example shown in Fig. 10, the link is to an object of type *IHTML*, shown in Fig. 9. The *CLSID* field is used to specify a class library containing executable code implementing the class specified in the *IID* field. For DCOM aficionados, this is used to find by a class moniker to locate the class object. The *URL* field specifies the location of the data to which the link refers.

The last field is used to specify the time at which the binding is resolved. There are currently two options supported: compile time and run time. These settings change the behaviour of the pre-processor and cause different code to be generated. When the compile-time option is chosen the pre-processor pre-fetches a copy of the target and stores it locally. In this case the code generated contains fewer run-time checks since the data will always be accessible. When run-time binding is employed, failure at run-time is possible and consequently the generated code needs to be more sophisticated. The code generated for the example program shown in Fig. 8 is given in the next section.



Location: http://www.cs.stir.ac.uk/cgi-

Object Creation/Modification: obj–23

Object Name:	telephonedirectory
Object Type:	IHTML
IID:	IID_IHTML
CLSID:	CLSID_CWebObject
URL:	www.dcs.gla.ac.uk/contact/index.html
Bind Time:	Construction Time

Fig. 10. Entering Details into the Binder

Binding Times and Errors. The code generated depends on the binding time specified in the binder. Fig. 11 shows a slightly simplified version of the code generated for the hyper-program shown in Fig. 8 if construction time (eager) binding is specified. This code assumes that the binder has loaded the Web page into the local cache (home/sag/cache). The dynamic case is similar but requires additional code to fetch the page across the network. The code generated is straightforward DCOM code.

```
void main(int argc, char** argv)
{
   OLECHAR *line = 0;
   IHTML* h = 0;
   IClassFactory *pcf = 0;
   HRESULT res = S_OK;
   IMoniker *pmk = 0;
   IBindCtx *pbc = 0;
   Check(CreateBindCtx(0,&pbc), “CreateBindCtx failed”);
   Check(CreateClassMoniker (CLSID_CWebObject, &pmk),
         “CreateClassMoniker failed”);
   Check(BindToObject(pbc,0,IID_IClassFactory,
         (void**)&pcf), “BindToObject failed”);
   Check(pcf->CreateInstance(0, IID_IHTML, (void**)&h),
         “Create Instance failed”);
   Check(h->openURL("/home/sag/cache/www.dcs.gla.ac.uk/
             contact/index.html"), “Open URL failed”);
   BOOL end = FALSE;
   BOOL is_found = FALSE;
   while(SUCCEEDED(h->at_end(&end)) && ! end) {
      if((SUCCEEDED(h->find_in_line(
      argv[1,&line,&is_found)) && is_found)) {
         printf( “Details are %s \n\r”, line ); break; }
      if( FAILED(h->next_line()) ) break;
   }
   if( end ) printf( “didn’t find %s", argv[1]);
   h->Release();
   pcf->Release();
}
```

Fig. 11. Simplified DCOM code generated for Fig. 8

Future Directions. All the examples and screen shots discussed this far describe a system that has been implemented at the University of Stirling. However, this code represents the start rather than the end-point of what we are trying to achieve. We stated earlier that we were seeking an integration of C++/DCOM with hyper-programming and the ideas embodied in the Hippo system. We now describe how we can use what we have implemented to date to achieve this.

```
void main (char** argv, int argc)
{
   BOOL end;
   IPersonSet *s = <(telephonedirectory)>;

   while(SUCCEEDED(s->at_end(&end)) && !end) {

      Person person;
      if(SUCCEEDED(s->next_person(&person) &&
      !strcmp(person.name,argv[1]))) {
         printf("Telephone number of %s is %s\n",
             argv[1],person.phone_no);
         break;
      }
   }

   if (end) printf("didn't find %s\n",argv[1]);
}
```

Fig. 12. A Strongly Typed C++ Hyper-Program

The program shown in Fig. 8 treats the Web data as an HTML file not as a typed entity. We would like to be able to re-write the hyper-program as shown in Fig. 12. In this example, rather than treating the data as HTML text, we have typed it as a set of objects of type *Person*. This requires a number of refinements to the mechanisms already implemented. First the HTML file must be typed as a set of *Person*. To achieve this, a MIDL interface definition of a set of *Person* is created as shown in Fig. 13. This type is structurally similar to the *IHTML* interface given earlier with the *line* type being replaced with records of type *Person*. Since the *IPersonSet* interface inherits from *IHTML*, it may use the *IHTML* interface to assist in the extraction of records of type *Person* from the text file.

```
typedef struct { OLECHAR *name; OLECHAR *phone_no;
                 OLECHAR *nickname; } Person;

interface IPersonSet : IHTML
{
   HRESULT next_person([out] Person* current );
}
```

Fig. 13. MIDL Definition of Person Set Interface

Some mechanism must be provided to convert the textual data retrieved over the Web into typed objects (in this case of type *Person*). This task is encoded in the library providing the implementation of *IPersonSet*. Whilst this implementation could be hand coded, a more desirable approach would be to generate it automatically from a specification. There are two basic approaches to this: (i) use the MIDL as a specification for the Web format and (ii) use the Web format as a specification to generate the MIDL.

If the first approach were employed, a tool could be engineered which took the MIDL interface and a URL as parameters and attempted to find records of the appropriate type in the file. In the case of the URL used in the examples in this Section, the fields are all comma separated making this task easy. This is similar to the construction of indices in database systems and the importation of records using Wizards in Microsoft Excel and Access. Once the index was created, generic code could be used to traverse the data and return records each time *next_person* was called. An alternative approach is to generate the IDL from the Web source. This approach is particularly attractive if the Web source is encoded in a structured or semi-structured manner, for example, using XML [34]. In both cases, generic code needs to exist which may be specialised to operate over records of an appropriate type. This may be achieved using the parametric polymorphism provided by the implementation language or using tools such as those suggested by Sheard and Stemple [35] or Kirby [36].

4 Conclusions

Our original motivation for hyper-programming was to allow the user to compose programs interactively, navigating the environment and selecting data items, including code, to be incorporated into the programs. We further believed that programming language systems could provide better support for the software engineering process than they do at present, in particular, with regard to the traditional *compose-compile-link-execute* cycle of program development.

From our early implementations of hyper-programming we summarised that the attendant benefits of the concept are:

- being able to perform program checking early
- support for source representations of all object closures
- being able to enforce associations from executable programs to source programs
- availability of an increased range of linking times
- increased program succinctness
- increased ease of program composition

Here we have developed the hyper-programming notion to presenting a single representation of data and code throughout the software process using hyper-code. Furthermore we have explored techniques for adapting hyper-programming to persistent contexts that do not enforce referential integrity, such as the WWW; and implementing and using hyper-programming in standardised languages and inter-operability mechanisms, such as C++ and DCOM.

5 Acknowledgements

We wish to acknowledge the work of Vivienne Dunstan on hyper-code and her contribution to the concepts presented here. The outline of the global hyper-programming challenges described in Section 3.5 relies heavily upon work done within the Hippo project by: David Lievens, Paolo Manghi, Steve Neely, Keith Sibson, Fabio Simeoni and Anna Stavrianou.

We also acknowledge the support of EPSRC under grant GR/L32699 “Compliant Systems Architecture” and the EC Working Group Pastel EC22552.

References

1. Morrison R., Connor R.C.H., Cutts Q.I., Dunstan V.S., Kirby G.N.C. Exploiting Persistent Linkage in Software Engineering Environments. Comp. J. 1995; 38,1:1-16
2. Metrowerks Inc. CodeWarrior Pro 5, 1999
3. 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: A. Albano and R. Morrison (ed) Persistent Object Systems, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy. Springer-Verlag, 1992, pp 376-393
4. Kirby G.N.C., Connor R.C.H., Cutts Q.I., Dearle A., Farkas A.M., Morrison R. Persistent Hyper-Programs. In: A. Albano and R. Morrison (ed) Persistent Object Systems, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy. Springer-Verlag, 1992, pp 86-106
5. Morrison R., Brown A.L., Connor R.C.H., Cutts Q.I., Dearle A., Kirby G.N.C., Munro D.S. Napier88 Reference Manual (Release 2.2.1). University of St Andrews, 1996
6. Atkinson M.P., Daynès L., Jordan M.J., Printezis T., Spence S. An Orthogonally Persistent Java™. ACM SIGMOD Record 1996; 25,4:68-75
7. Cutts Q.I., Connor R.C.H., Kirby G.N.C., Morrison R. An Execution Driven Approach to Code Optimisation. In: Proc. 17th Australasian Computer Science Conference (ACSC'94), Christchurch, New Zealand, 1994, pp 83-92
8. Stroustrup B. The C++ Programming Language. Addison-Wesley, 1986

9. The Common Object Request Broker: Architecture and Specification, Revision 2.2. Object Management Group (OMG), 1998
10. Microsoft Corporation. DCOM Technical Overview. , 1996
11. Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott W.P., Morrison R. An Approach to Persistent Programming. Comp. J. 1983; 26,4:360-365
12. Atkinson M.P., Morrison R. Procedures as Persistent Data Objects. ACM ToPLaS 1985; 7,4:539-559
13. Goldberg A., Robson D. Smalltalk-80: The Language and its Implementation. Addison Wesley, Reading, Massachusetts, 1983
14. O'Brien P.D., Halbert D.C., Kilian M.F. The Trellis Programming Environment. ACM SIGPLAN Notices 1987; 22,12:91-102
15. Dearle A., Brown A.L. Safe Browsing in a Strongly Typed Persistent Environment. Comp. J. 1988; 31,6:540-544
16. Bretl B., Maier D., Otis A., Penney J., Schuchardt B., Stein J., Williams E.H., Williams M. The GemStone Data Management System. In: W. Kim and F. Lochovsky (ed) Object-Oriented Concepts, Databases and Applications. ACM Press and Addison Wesley, 1989, pp 283-308
17. Cooper R.L. On The Utilisation of Persistent Programming Environments. Ph.D. thesis, University of University of Glasgow, 1990
18. Kirby G.N.C., Dearle A. An Adaptive Graphical Browser for Napier88. University of St Andrews Report CS/90/16, 1990
19. Zirintsis E., Dunstan V.S., Kirby G.N.C., Morrison R. Hyper-Programming in Java. In: R. Morrison, M. Jordan and M. P. Atkinson (ed) Advances in Persistent Object Systems, Proc. 8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3), Tiburon, California, 1998. Morgan Kaufmann, 1999
20. Connor R.C.H., Cutts Q.I., Kirby G.N.C., Moore V.S., Morrison R. Unifying Interaction with Persistent Data and Program. In: P. Sawyer (ed) Interfaces to Database Systems, Proc. 2nd International Workshop on User Interfaces to Databases, Ambleside, Cumbria, 1994. Springer-Verlag, 1994, pp 197-212
21. Kirby G.N.C. Reflection and Hyper-Programming in Persistent Programming Systems. Ph.D. thesis, University of University of St Andrews, 1992
22. Morrison R., Balasubramaniam D., Greenwood M., Kirby G.N.C., Mayes K., Munro D.S., Warboys B.C. ProcessBase Reference Manual (Version 1.0.4). Universities of St Andrews and Manchester, 1999
23. Morrison R., Balasubramaniam D., Greenwood M., Kirby G.N.C., Mayes K., Munro D.S., Warboys B.C. A Compliant Persistent Architecture. To Appear: Proc. Software—Practice and Experience, 1999
24. Connor R.C.H., Cutts Q.I., Kirby G.N.C., Morrison R. Using Persistence Technology to Control Schema Evolution. In: Proc. 9th ACM Symposium on Applied Computing, Phoenix, Arizona, 1994, pp 441-446
25. Connor R.C.H., Atkinson M.P., Berman S., Cutts Q.I., Kirby G.N.C., Morrison R. The Joy of Sets. In: C. Beeri, A. Ohori and D. E. Shasha (ed) Database Programming Languages, Proc. 4th International Conference on Database Programming Languages (DBPL4), New York City. Springer-Verlag, 1993, pp 417-433
26. Atkinson M.P., 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
27. Stemple D., Fegaras L., Sheard T., Socorro A. Exceeding the Limits of Polymorphism in Database Programming Languages. In: F. Bancilhon, C. Thanos and D. Tsichritzis (ed) Lecture Notes in Computer Science 416, Proc. 2nd International Conference on Extending Database Technology (EDBT'90), Venice, Italy. Springer-Verlag, 1990, pp 269-285

28. 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. To Appear: Proc. The FIDE Book, 1999
29. Connor R.C.H., Sibson K., Manghi P. On the Unification of Persistent Programming and the World-Wide Web (LNCS). In: Lecture Notes in Computer Science, Proc. Workshop on the Web and Databases (WebDB'98), Valencia, Spain. Springer-Verlag, 1998
30. Simeoni F. Extracting Typed Data from Semi-Structured Collections. MSc thesis, University of University of Glasgow, 1998
31. Connor R.C.H., Manghi P., Simeoni F. A Kinded Approach to Extracting Typed Subsets from Semi-Structured Data. *in preparation*. Please contact the authors.
32. Stallman R.M. EMACS: The Extensible, Customizable Self-Documenting Display Editor. ACM SIGPLAN Notices 1981; 16,6:147-156
33. Altrasoft. Hyperbole , 1998
34. Bray T., Paoli J., Sperberg-McQueen C.M. Extensible Markup Language (XML) 1.0. W3C, 1998
35. Stemple D., Sheard T., Fegaras L. Linguistic Reflection: A Bridge from Programming to Database Languages. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 844-855
36. Kirby G.N.C., Connor R.C.H., Morrison R. START: A Linguistic Reflection Tool Using Hyper-Program Technology. In: M. P. Atkinson, D. Maier and V. Benzaken (ed) Persistent Object Systems, Proc. 6th International Workshop on Persistent Object Systems (POS6), Tarascon, France. Springer-Verlag, 1994, pp 355-373