

This paper should be referenced as:

Morrison, R., Connor, R.C.H., Kirby, G.N.C. & Munro, D.S. "Can Java Persist?". In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996) pp 34-41.

Can Java Persist?

Ron Morrison, Richard Connor, Graham Kirby and David Munro

School of Mathematical and Computational Sciences,
University of St Andrews, St Andrews, Fife, Scotland KY16 9SS

{ron, richard, graham, dave}@dcs.st-and.ac.uk

Abstract

This paper briefly and selectively reviews the experience gained in designing and implementing the orthogonally persistent programming languages PS-algol and Napier88. A major design issue is how much is built into the support system and how much is built on top of the language itself. The PS-algol and Napier88 systems provide at the system or language level: a persistent store with root(s), reachability and referential integrity; code as data; an infinite union type with dynamic injection and projection; and two type *magic* procedures, one to find a type representation of a value, and one to convert a sequence of bytes into a language value. Within a strongly typed system this provision allows the programming techniques of strongly typed linguistic reflection, hyper-programming and persistent schema evolution, the last two of which are especially significant since they are new paradigms that are only available in orthogonally persistent systems. The essence of the paper is to suggest how the above facilities may be provided in Java to enable these new persistent programming paradigms.

1 Introduction

In the early days of persistence research, it was hypothesised "that it should be possible to add persistence to an existing language with minimal change to the language" [ABC+83]. While the statement turned out to be true, it was not true in the manner that was first expected. That is, "an existing language" turned out to be existentially quantified rather than the intended universal quantification. The subtlety is that to support persistence the language environment must provide a basic number of facilities either within the language itself or within its run-time support system. Since then many persistent languages have been developed including Abstract Data Store [Pow85], Amber [Car86], Fibonacci [ABG+93], Flex [Cur85], Galileo [ACO85], Napier88 [MBC+94], PS-algol [PS88], TI Persistent Memory System [Tha86], Trellis/Owl [SCW85] and Tycoon [MS92].

Here we selectively draw on the PS-algol/Napier88 experience which developed a strongly typed orthogonally persistent programming environment. We show how, by providing a basic number of facilities at the system support level, the programming techniques of strongly typed linguistic reflection [SSS+92], hyper-programming [KCC+92] and persistent schema evolution [CAB+93] may be programmed within the language. The last two of these are especially significant since they are new paradigms that are only achievable in orthogonally persistent systems. The essence of the paper is to demonstrate how these basic facilities may be provided in Java with the same attendant benefits.

2 The Joys of Persistence

2.1 PS-algol (Mk 1)

The first successful attempt to provide orthogonal persistence was PS-algol (Mk 1) [ACC82]. This took the language S-algol [Mor79] and added a persistent store. The language had a small number of base types *int*, *real*, *bool*, *string* and *picture* as well as two constructor types *vector* and *structure*. Thus, the universe of discourse was infinite since the constructors could contain any types, by the principle of type completeness [Str67], and therefore graphs of any complexity could be constructed and stored.

Programs were prepared and compiled outside the persistent store and could access the roots of persistent objects through databases which were implemented at the top level by directory structures called tables. Figure 1 shows a number of programs accessing the database directories in the persistent store and, by following pointers, the objects within the stored graphs.

The problem of how to identify which objects persisted after program execution was alleviated by the fact that S-algol was garbage collected. This easily extended to persistence by reachability which is now the common method employed by others.

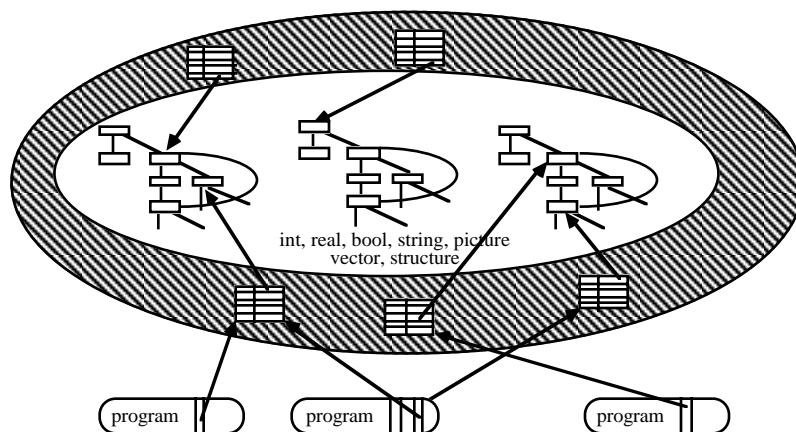


Figure 1: PS-algol (Mk 1) Persistent Store

The use of S-algol as a starting point was not, however, a random choice since it supports an infinite union type of all structures, called **pntr**¹. This allows the directories to return **pntr** types when indexed by a name and for this type to be projected onto the particular type that was asserted by the program and checked by the compiler. Thus, this dynamic bind and type check allowed the rest of the program using that type to be statically checked. Furthermore it allowed the data and programs to be evolved separately since these points of dynamic checking could be extended to the object graph. Programs accessing the data needed only to specify the part of the graph of interest accurately.

In Figure 1, the shaded area depicts the points at which the programs dynamically bind to the persistent store through the database directories.

The advances of PS-algol (Mk 1) were a persistent store with roots and reachability together with an infinite union type supporting dynamic injection and projection operations. While this huge advance was very exciting in 1983, and indeed is the model of persistence presented by many language/OODBMS systems today, e.g. [Deu91], it soon became clear that the next step was to remove the artificial difference between code and data and put the code within the persistent environment.

2.2 PS-algol (Mk 2)

PS-algol (Mk 2) added higher order procedures to its universe of discourse [AM85]. The procedures were first class and could be assigned to variables, nested, passed as parameters and returned as results of other procedures. This allowed programming of modules, abstract datatypes, partial application, views, code libraries, and separate compilation.

¹ It was also serendipitous that the S-algol implementation technology was so amenable to persistent overtures. Earlier attempts with Pascal and algol 68 were less successful.

Initially the programs were composed and compiled outside the store. When they were run, however, it was possible to assign a procedure to a location within a persistent data structure and thus over time code and data were freely intermixed in the persistent graphs. This yielded a single view of the total computation space, a concept which recurs throughout this research. Given that the procedures were now within the store it was also possible for them to access data without going through the directory structure. For example, a procedure having accessed a data object through the database route could then store the pointer to the object within its closure for future use. This was exploited to increase static binding within the system and is the basis of hyper-programming which was developed later.

Realising that the compiler, which was written in PS-algol, was a procedure and could be placed in the store yielded the ability to prepare, compile and run programs completely within the persistent environment. The PS-algol compiler type is given in Figure 2.

```

structure rons.proc (proc (string -> int) rons)
! This declares a structure class (type) called rons.proc which has one field called rons
! which is a procedure that takes a string parameter and yields an integer result

let compiler = proc (string source ; pntr proc.holder -> pntr)
! The compiler is a procedure that takes the source as a string and a value of a structure
! class. It returns the compiled source as a procedure wrapped up in the structure class

```

Figure 2: PS-algol Compiler Type

The compiler compiles the procedure found in *source* and type checks it against the single procedure field of *proc.holder*. If successful a structure is returned containing the compiled procedure. The structure will be the same structure class as *proc.holder* and therefore may be projected by the user using this definition. If the compilation fails a linked list of error messages is returned.

For the compiler to be written in PS-algol it must have access to two system *magic* functions. The first takes an object and returns a description of its type. This is required for type checking the procedure type that is supplied and for ensuring that the compiler and the program calling it share the same structure class definition while not in the same scope. Thus in Figure 2, the compiler can produce a value that has the structure class *rons.proc* which can be used by the calling program. The second function takes a sequence of characters that have been produced by the compiler and performs a type coercion so that they may now be considered as a procedure value. For convenience we will call these *typeOf* and *coerceToValue* respectively.

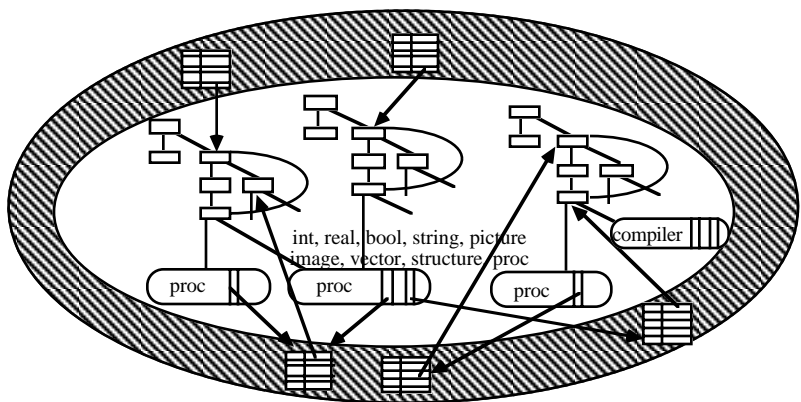


Figure 3: PS-algol (Mk 2) Persistent Store

Figure 3 depicts the PS-algol (Mk 2) persistent store with the compiler as a persistent data object.

The presence of the compiler within the persistent environment provides the ability for the programming technique of linguistic reflection [SSS+92] which is defined as the ability of a running program to generate new program fragments and to integrate these into its own execution. The advance of PS-algol (Mk 2) is the integration of this style of reflection with a compiled, strongly typed language.

An example of such a linguistic reflective program is a persistent object store browser [DB88, DCK90] which displays a graphical representation of any value presented to it. The browser may encounter values in the persistent store for which it does not have a static type description. This may occur, for example, for values that are added to the store after the time of definition of the browser program. For the program to be able to denote such values, they must belong to an infinite union type, such as **pntr**.

Before any operations may be performed on a value of an infinite union type it must be projected onto another type with more type information. This projection typically takes the form of a dynamic check of the value's type against a static type assertion made in the program that uses it. The browser program takes as parameter an infinite union type, a **pntr**, to allow it to deal with values whose types were not predicted at the time of implementation. However the program cannot contain static type assertions for all the types that may be encountered as their number is unbounded. There are two possibilities for the construction of such a program: it may either be written in a lower-level technology [KD90] or else be written using linguistic reflection.

To allow a reflective solution the program must be able to discover dynamically the specific type of a value of the union type. Such functionality may be provided by one of the two magic functions defined above, *typeOf*.

The generic browser takes a specification of the object type and generates the source of a procedure to browse over it. This procedure is compiled and may be applied to perform the browsing. The browser then stores the procedure in the persistent store for future use should it encounter an object of the same type. The browser repeats this action for every new type it discovers.

All the system functionality required to invent the new persistent programming paradigms of hyper-programming and persistent schema evolution was now available. Historically, they had to wait until Napier88 was invented, designed and implemented. We will return to that story and to the joys of hyper-programming and persistent schema evolution. For the present we will concentrate on how the functionality to support these techniques may be provided in Java.

3 Persistence in Java

The joys of persistence can be delivered by building the mechanisms into the interpreter or building on top of the language facilities. The PS-algol/Napier88 experience built the programming environment on top of the language but required the following support within the system:

- an infinite union type with injection and projection operations using structural type equivalence for separately prepared programs and data
- the compiler as a persistent object, with functions *coerceToValue* and *typeOf*, for linguistic reflection
- a persistent store with root(s), reachability and referential integrity
- code as data, including the ability to store references to objects in the code

We now turn our attention to providing these facilities in Java in order to obtain the same benefits as the PS-algol/Napier88 experience.

3.1 An Infinite Union Type

The class *Object* is an infinite union type. All objects have this class and injection is implicit, however, care should be taken to use wrappers with primitive classes. Projection is by casting onto a subclass, for example:

```
Object ron = ...  
Person me = (Person) ron           // Coerce (project, cast) the value onto the class  
Person
```

For this to work in Java's explicit single inheritance hierarchy, the exact class of *ron* must be the same as or an extended class of *Person*. If the cast is not valid a *ClassCastException* is thrown. In order to avoid handling exceptions a cast can be tested before being executed. If it is valid true is returned and false otherwise. For example

```
if (ron instanceof Person) {       // Would return true if the exact class of ron is the  
                                   // same as or an extended class of Person.
```

The *null* reference, which is not an instance of any class, will always result in a false result in an *instanceof* test.

The class *Object* can therefore be used in a manner similar to that of the types **pntr** in PS-algol and **env** and **any** in Napier88.

Java uses name equivalence rather than structural equivalence. A discussion of the trade-offs in these mechanisms in persistent systems is given in [CBC+90].

3.2 Type Magic and a Persistent Compiler

Each Java class has a unique object of class which is a partial description of the class. Arrays share class objects with others of the same element class and number of dimensions. The definition of *Object* provides a method to obtain the class object.

```
public class Object {  
    public final Class getClass ();  
    ...  
}
```

The class *Class* provides one of the basic needs for **linguistic reflection**. It is defined in Figure 4.

```

public final class Class {
    public String toString ();
    public String getName ();
    public boolean isInterface ();
    public Class getSuperclass ();
    public Class[] getInterfaces ();
    public Object newInstance()
        throws InstantiationException, IllegalAccessException;
    public ClassLoader getClassLoader ();
    public static Class forName (String className)
        throws ClassNotFoundException;
}

```

Figure 4: The class Class

Instances of the class *Class* represent the class in such a way that it can be manipulated by a running Java program. Objects of class *Class* cannot be created by the user since there is no public constructor for the class. Instead the Java Virtual Machine constructs such objects when classes are loaded.

The description of the class given by the class object is only partial since it does not include the class description of the class members. It is clear, however, that such functionality would not endanger the class security of Java since it is a representation that is being returned rather than the value itself. The member descriptions are required to write linguistically reflective programs such as the browser described above. Thus we propose an additional method for the class *Class* called *getMembers* which is defined as follows:

```

public String[] getMembers ();

```

The *getMembers* method returns an array of strings, one for each member, that describes the access specifiers, return and parameter class names for each method in the class and the access specifiers and class names for each field. Given this the *typeOf* magic function can now be implemented as follows:

- The *getClass* method of the class *Object* returns an object of the class *Class*. This class has methods to decompose the structure of the class representation, for example, *getName*, *getSuperClass*, *getInterfaces*. This can also be found from its name using *Class.forName* (“myClass”). The *getMembers* method returns the description of the class under inspection. The *getSuperClass* method returns the class object for the superclass. It is therefore possible to recursively construct the complete description of the object.

Once the functionality of *typeOf* can be implemented it is possible to write linguistically reflective programs within the persistent environment, given that all the other functionality described in this section is also in place. We now turn our attention to making the compiler a persistent object for which we require, in addition to *typeOf*, the *coerceToValue* function.

Central to making the compiler a persistent object is the mechanism for loading classes. Each class provides its own class loader which can be obtained from the class object using the *getClassLoader* method. The class loader class is defined in Figure 5.

```

public abstract class ClassLoader {
    protected ClassLoader () throws SecurityException;
    protected abstract Class loadClass (String name, boolean resolve)
        throws ClassNotFoundException;
    protected final Class defineClass (byte[] data, int offset, int length)
        throws NullPointerException, IndexOutOfBoundsException, ClassFormatError;
    protected final void resolveClass (Class c) throws NullPointerException;
    protected final Class findSystemClass (String name) throws
ClassLoaderNotFoundException;
}

```

Figure 5: The class ClassLoader

It should be noted that three of the *ClassLoader* methods are **final** and therefore may not be overridden in an extended class and that one method is **abstract**. At present the compiler produces a class file for each class which is then loaded using the class loader. During the loading of the class the interpreter creates an object of class *Class* for the class.

An object of class *Class* can be created from a sequence of bytes by the *defineClass* method of the *ClassLoader* class. For example,

```

byte[] buffer = ...; // These are read from the file system
Class newClass = defineClass (buffer, 0, buffer.length);

```

The *defineClass* method ensures that the class defined in the buffer is in the correct format and is therefore a valid Java class. It provides the functionality of *coerceToValue*. Given this it is now possible to define a compiler class, an instance of which could reside in the persistent store.

```

class Compiler { // A run time callable compiler
    public Class compile (Source source) { ...

```

Thus the *compile* method takes some source code and produces an instance of the class *Class*. Inside the compiler the *defineClass* method must be used to invoke the type magic by writing the compiled code into a class file and using *defineClass* to load it and coerce it to the correct class. The compiler object is now no different from any other Java object and may be placed in the persistent store. The compiler would use its own instantiation of *ClassLoader* to resolve the file names that it uses.

The use of files in a persistent environment is somewhat unambitious. However we do not wish to see the class security compromised and we cannot overload *defineClass* since it is declared **final**. To avoid the use of files we define a method in the *Compiler* class called *compileToBytes* which takes the source and returns a byte sequence which is the code for the compiled class.

```

class Compiler { // A run time callable compiler
    public Class compile (Source source) { ...
    private byte[] compileToBytes (Source source) { ...

```


The *compile* method calls the *compileToBytes* method to obtain the compiled code. It now uses its own version of the class loader to load the class and return a value of class *Class*. Figure 6 outlines the definition of a class loader for the compiler.

```
public abstract class ClassLoader {
    protected abstract Class loadClass (String name, ...
                                     // This is the definition in ClassLoader
    ...

class CompilerLoader extends ClassLoader { ...
    public Class loadClass (byte[] compiledClass) {
        result = defineClass (compiledClass, 0, compiledClass.length);...
    ...
// Within the compile method the code to use this would be
CompilerLoader loader = new CompilerLoader ();
Class result = loader.loadClass (compileToBytes (source));

// and to create an instance of a compiled class
MyType me = (MyType) compile (some source).newInstance (); // create an instance
...

```

Figure 6: Loading the Compiled Java Code

A class loader is defined by extending the abstract *ClassLoader* class and providing an implementation for its *loadClass* method. The key difference is that it works over a given sequence of bytes rather than reading bytes from a file. In this case, the concrete class is called *CompilerLoader* and the *loadClass* method is overloaded to enable it use the byte sequence provided by the *compileToBytes* method.

3.3 The Persistent Store

PJava₀ uses the directory method similar to PS-algol:

- Every PJava₀ store has an object of class *PJavaStore* which is available to Java programs and has code to manipulate the persistent store
- It has a method *registerPRoot (aName, anObject)*
- All values reachable from *anObject* will be preserved
- These values can be found by subsequent programs using *getPRoot (aName)*

```
Object ron = getPRoot ("rons")    // Get the persistent value stored using the index rons

Person me = (Person) ron         // Coerce (project, cast) the value onto the class
Person

```

Reachability and referential integrity are obtained bin PJava₀ by observing the following rules [AJD+96]. For reachability these are that any non-persistent class X will be promoted to the persistent store if:

- 1 class X is directly referenced from a persistent object or from an object that has already been promoted,
- 2 an instance of the class X is being promoted to the persistent store for the first time,
- 3 class X is a superclass of the class being promoted,

- 4 an instance of class X is a static or instance variable of the class being promoted,
- 5 an instance of class X is a parameter of a method of a class being promoted,
- 6 an instance of class X is a variable of a method of the class being promoted, or
- 7 a static variable of class X is used by the class being promoted.”

The top level directory structure of PS-algol and PJava₀ is not the only possible mechanism. An alternative technique, used in Napier88, is for each persistent store to be a value of class *Object* which can be cast onto the particular class for use. The advantage is that each store can have its own structure without imposing any predefined system wide rules. Thus stores can develop independently. Programs accessing the stores can discover the structure by using the linguistic reflection programming technique shown above.

3.4 Code as Data and System Evolution

This is the last piece in the jigsaw and perhaps the most difficult since the underlying philosophy of Java is that of code libraries and files of data for persistent storage. Objects are updateable in Java and are part of the computational universe of discourse but the code (methods), which belongs to the classes, are not. Thus, whereas objects may be placed in the persistent store and freely manipulated, passed around and updated, classes and methods are quite a different prospect.

The object graph can accommodate arbitrary structural change. However the class hierarchy, once defined, can only be extended. This makes it impossible to insert an extended class between two others in the hierarchy even although, since the system has evolved, it would be safe and advantageous to do so. Furthermore it is also impossible to alter a class definition or to update the code in initialisers and methods, once they are defined, in order to repair a bug or implement a smarter algorithm.

The fall back position is to completely rebuild the persistent store in order to accommodate changes to classes or methods.

Given that there is a security issue about who is allowed to make changes, it should be possible to update a class or the hierarchy in a manner that is class compatible with the old definitions without disturbing the semantics of the system. For this we propose that the class *Class* provides an *updateClass* method which checks the user's authority to make the change and the class compatibility for the change. Any alteration must leave the class in the same position relative to its superclasses and subclasses. Changes to the hierarchy require the specification of two class objects between which the new class has to be inserted.

These changes to the class hierarchy do not accommodate arbitrary alteration. An alternative is to use the schema evolution method developed in Napier88 and described in the next section. The schema evolution technique utilises hyper-programming and given the basic facilities described above, it should be possible to programme both persistent programming techniques in Java.

4 Applications of Persistence Technology

The contribution of Napier88 is to provide a type system that more accurately describes the computations required for persistent programming. This aids the programming of the programming environment in Napier88 itself [KBC+96]. The facilities of Napier88 relevant to this discussion are:

- graphical data types picture and image
- constructors: structure (x), variant (+), and procedure (->)
- type completeness

- parametric polymorphism (genericity)
- existentially quantified types (views and information hiding)
- infinite unions **any** and **env** (dynamic binding and type checking)
- collections of bindings for name space control and incremental system evolution
- a strongly typed populated store

Figure 7 depicts the Napier88 persistent store in which all the computation is contained within the store. Notice that the root of persistence is now a single objects of type **any** and not directory structures. Given such a system it is now possible to invent hyper-programming.

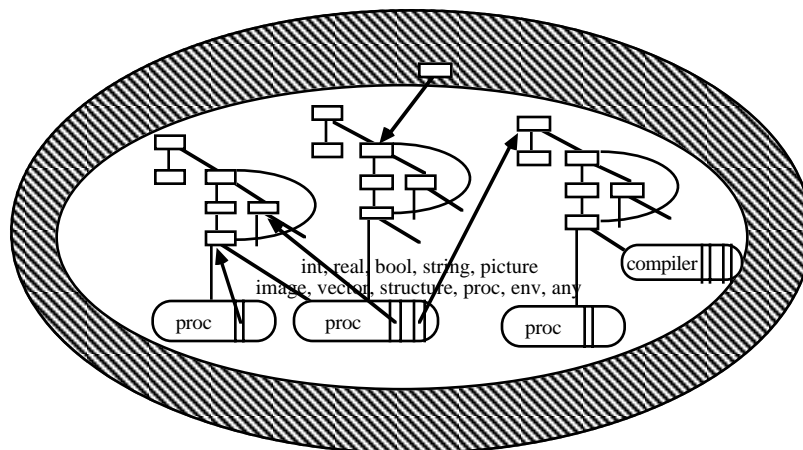


Figure 7: Napier88 Persistent Store

The presence of a persistent environment in which programs are composed, compiled, linked and run means that persistent objects can be linked into the program source, instead of having the more traditional textual descriptions of where to find persistent values. The source text now contains some text and some links to persistent values and as such it is a non-flat representation of the program. By analogy with hyper-text, a program containing both text and links to persistent values is called a *hyper-program* [KCC+92].

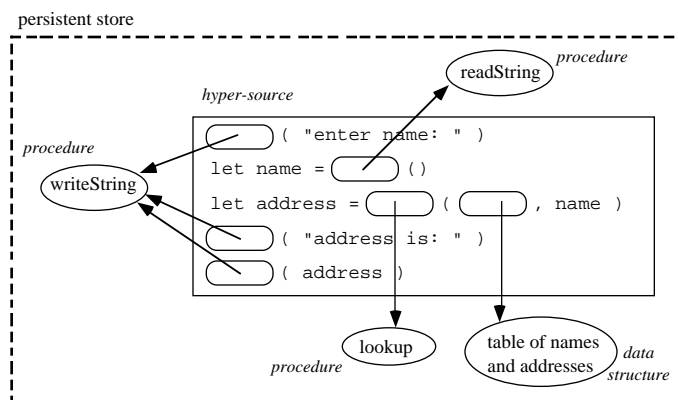


Figure 8: A Hyper-program

Figure 8, taken from [KCC+92], shows an example of a hyper-program. The first link is to a 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 look up the address in a table data structure linked into the hyper-program. The address is then written out. Note that code objects (*readString*, *writeString* and *lookup*) are denoted using exactly the same mechanism

as data objects (the table). Note also that the object names used in this description have been associated with the objects for clarity only, and are not part of the semantics of the hyper-program.

Figure 9, again taken from [KCC+92], shows an example of the user interface which might be presented to the programmer by a hyper-program editing tool. The editor contains embedded light-buttons representing the hyper-program links; when a button is pressed the corresponding object is displayed in a browser window. The browser is also used to select persistent objects for linking into hyper-programs under construction.

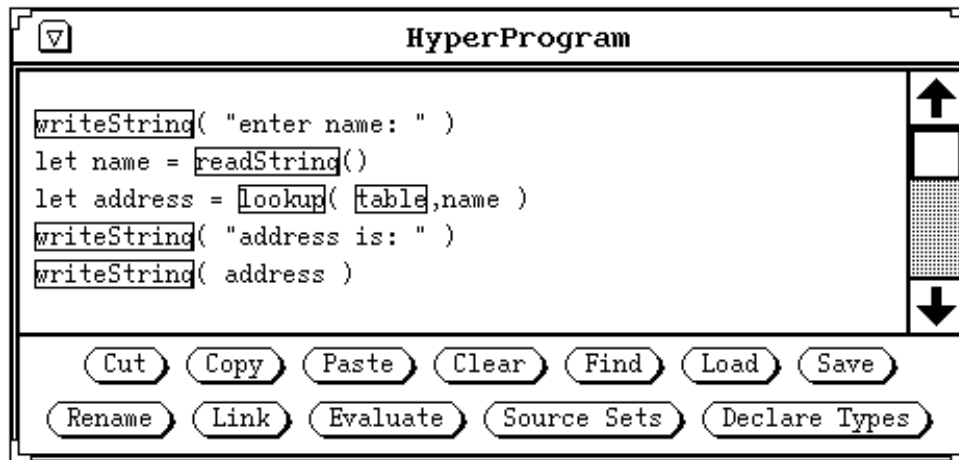


Figure 9: User Interface to a Hyper-program Editor

The benefits of hyper-programming are discussed in [FDK+92, Kir92, KCC+92] and include:

- being able to perform program checking early — access path checking and type checking for linked components may be performed during program construction;
- being able to enforce associations from executable programs to source programs — links between source and compiled versions may be used;
- support for source representations of all procedure closures—free variables in closures may be represented by links, thus allowing hyper-programs to be used for both source and run-time representations of programs; and
- increased program succinctness—access path information, specifying how a component is located in the environment, may be elided.

The non-flat nature of the hyper-program allows a single program representation, the hyper-program, to be presented to the programmer at all stages of the software development process. In constructing a program, the programmer writes hyper-programs. 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-program 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 artefacts of how the program is stored and executed, and as such are completely hidden from the programmer. Only one set of tools is required for manipulating the hyper-program thereby simplifying the user interface and the complexity of the system implementation. This permits concentration on the inherent complexity of the application rather than on that of the support system.

An outline of how hyper-programs may be used to increase the effectiveness of components in software environments such as version control, configuration management and documentation systems is given in [MCC+95]. Furthermore, the same advantages accrue to

other activities supported by software environments such as debugging, profiling and optimisation [Cut92].

Achieving the full advantage of a hyper-programming system in the Java context requires further research. However one approach is to consider the *getMember* method which could return for each member a hyper-program representation which was a mixture of text and pointers for the language elements such as types and code. This forms the basis for constructing hyper-programs around which the rest of the system can be built.

Since all the computation takes place within the persistent environment, it is possible to evolve the system while it is executing. Changes to program and data with the invariant of fixed meta-data are normally handled by updates to procedures and data respectively. The difficult problem is to change the meta-data while keeping all the existing programs and data consistent with the semantics of the change [CCK+94].

The persistent environment can provide an underlying technology which allows a schema editor to locate and change, either manually or automatically, all affected program and data. The advantages of the mechanism are that it provides understandable semantics for evolution by controlling when the changes are made and by ensuring that changes to schema, program and data are consistent and made in lock step. Furthermore, these changes may be grouped together as a transaction within a live system; the accommodation of lazy data changes allows minimum loss of availability.

As an example, consider that 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 may thus transform the programs and data affected by a schema edit. This is achieved as follows:

1. Locate, from the schema, all affected programs and data.
2. For each program which may be affected, obtain its hyper-program.
3. 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.
4. Update the old program with the new one.
5. 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-program 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 the strategy possible. Transferring this technology to Java is dependent on providing the basic facilities defined above.

5 Conclusions

From the PS-algol/Napier88 experience we have identified a number of basic facilities that are required for the provision of orthogonal persistence. These are: a persistent store with root(s), reachability and referential integrity; code as data; an infinite union type with dynamic injection and projection; and two type magic procedures, one to find a type

representation of a value, and one to convert a sequence of bytes into a language value. While this is not the only manner in which orthogonal persistence can be provided, the approach allows all the other infrastructure for the persistent programming paradigms of strongly typed linguistic reflection, hyper-programming and persistent schema evolution to be built within the language.

In this paper we have discussed how these basic facilities may be provided in Java with the same attendant benefits. We have proposed *getMembers* and *updateClass* methods of the class *Class*. The rest is coding ...

6 Acknowledgements

This work is partially supported by the EPSRC under grant GR/J 67611 "Delivering the Benefits of Persistence to System Construction". Our thanks also to Huw Evans for his insights into the Java implementation.

7 References

- [ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. "An Approach to Persistent Programming". *Computer Journal* 26, 4 (1983) pp 360-365. URL: <http://www.fide.dcs.st-and.ac.uk/Publications/1983.html#approach.persistence>.
- [ABG+93] Albano, A., Bergamini, R., Ghelli, G. & Orsini, R. "An Object Data Model with Roles". In Proc. 19th International Conference on Very Large Data Bases, Dublin, Ireland (1993) pp 39-51.
- [ACC82] Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. "PS-algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices* 17, 7 (1982) pp 24-31.
- [ACO85] Albano, A., Cardelli, L. & Orsini, R. "Galileo: a Strongly Typed, Interactive Conceptual Language". *ACM Transactions on Database Systems* 10, 2 (1985) pp 230-260.
- [AJD+96] Atkinson, M.P., Jordan, M.J., Daynès, L. & Spence, S. "Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System". In Proc. 7th International Workshop on Persistent Object Systems, Cape May, NJ, USA (1996).
- [AM85] Atkinson, M.P. & Morrison, R. "Procedures as Persistent Data Objects". *ACM Transactions on Programming Languages and Systems* 7, 4 (1985) pp 539-559.
- [CAB+93] Connor, R.C.H., Atkinson, M.P., Berman, S., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. "The Joy of Sets". In **Database Programming Languages**, Beerli, C., Ogori, A. & Shasha, D.E. (ed), Springer-Verlag, Proc. 4th International Conference on Database Programming Languages, New York City (1993) pp 417-433. URL: <http://www.fide.dcs.st-and.ac.uk/Publications/1993.html#joy>.
- [Car86] Cardelli, L. "Amber". In **Lecture Notes in Computer Science 242**, Springer-Verlag (1986) pp 21-47.
- [CBC+90] Connor, R.C.H., Brown, A.B., Cutts, Q.I., Dearle, A., Morrison, R. & Rosenberg, J. "Type Equivalence Checking in Persistent Object Systems". In **Implementing Persistent Object Bases, Principles and Practice**, Dearle, A., Shaw, G.M. & Zdonik, S.B. (ed), Morgan Kaufmann, Proc. 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA (1990) pp 151-164. URL: <http://www.fide.dcs.st-and.ac.uk/Publications/1990.html#type.equiv>.

- [CCK+94] 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. URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1994.html#schema.evolution>.
- [Cur85] Currie, I.F. "Filestore and Modes in Flex". In Proc. 1st International Workshop on Persistent Object Systems, Appin, Scotland (1985) pp 325-334.
- [Cut92] Cutts, Q.I. "Delivering the Benefits of Persistence to System Construction and Execution". Ph.D. Thesis, University of St Andrews (1992). URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1992.html#thesis.qc>.
- [DB88] Dearle, A. & Brown, A.L. "Safe Browsing in a Strongly Typed Persistent Environment". Computer Journal 31, 6 (1988) pp 540-544. URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1988.html#safe.browsing>.
- [DCK90] Dearle, A., Cutts, Q.I. & Kirby, G.N.C. "Browsing, Grazing and Nibbling Persistent Data Structures". In **Persistent Object Systems**, Rosenberg, J. & Koch, D.M. (ed), Springer-Verlag, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia (1990) pp 56-69. URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1990.html#browsing.nibbling>.
- [Deu91] Deux, O. "The O₂ System". Communications of the ACM 34, 10 (1991) pp 34-48.
- [FDK+92] Farkas, A.M., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R. & Connor, R.C.H. "Persistent Program Construction through Browsing and User Gesture with some Typing". In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 376-393. URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1992.html#browsing.gesture>.
- [KBC+96] Kirby, G.N.C., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Dunstan, V.S., Morrison, R. & Munro, D.S. "Napier88 Standard Library Reference Manual (Release 2.2.1)". University of St Andrews (1996). URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1996.html#napier.lib.man.221>.
- [KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. "Persistent Hyper-Programs". In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 86-106. URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1992.html#persistent.hyperprograms>.
- [KD90] Kirby, G.N.C. & Dearle, A. "An Adaptive Graphical Browser for Napier88". University of St Andrews Technical Report CS/90/16 (1990). URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1990.html#napier.browser>.
- [Kir92] Kirby, G.N.C. "Reflection and Hyper-Programming in Persistent Programming Systems". Ph.D. Thesis, University of St Andrews (1992). URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1992.html#thesis.gk>.
- [MBC+94] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. "The Napier88 Reference Manual (Release 2.0)". University of St Andrews Technical Report CS/94/8 (1994). URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1994.html#napier.ref.man.2>.
- [MCC+95] Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S. & Kirby, G.N.C. "Exploiting Persistent Linkage in Software Engineering Environments". Computer Journal 38, 1 (1995) pp 1-16. URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1995.html#exploiting.linkage>.
- [Mor79] Morrison, R. "S-algol Language Reference Manual". University of St Andrews Technical Report CS/79/1 (1979).

- [MS92] Matthes, F. & Schmidt, J.W. "Definition of the Tycoon Language TL - A Preliminary Report". University of Hamburg, Germany Technical Report FBI-HH-B-160/92 (1992).
- [Pow85] Powell, M.S. "Adding Programming Facilities to an Abstract Data Store". In Proc. 1st International Workshop on Persistent Object Systems, Appin, Scotland (1985) pp 139-160.
- [PS88] PS-algol "PS-algol Reference Manual, 4th edition". Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).
- [SCW85] Schaffert, C., Cooper, T. & Wilpot, C. "Trellis Object-Based Environment Language Reference Manual". DEC Systems Research Center (1985).
- [SSS+92] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. & Alagic, S. "Type-Safe Linguistic Reflection: A Generator Technology". ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/49 (1992). URL: <http://www-fide.dcs.st-and.ac.uk/Publications/1992.html#linguistic.reflection>.
- [Str67] Strachey, C. **Fundamental Concepts in Programming Languages**. Oxford University Press, Oxford (1967).
- [Tha86] Thatte, S.M. "Persistent Memory: A Storage Architecture for Object Oriented Database Systems". In Proc. ACM/IEEE International Workshop on Object-Oriented Database Systems, Pacific Grove, California (1986) pp 148-159.