Accepted for publication in *Microprocessors and Microsystems* published by Elsevier.

Mechanisms for Controlling Evolution in Persistent Object Systems

R. Morrison, R.C.H. Connor, Q.I. Cutts, G. Kirby & D. Stemple[†]

Department of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland

[†] Department of Computer Science,
University of Massachusetts, Amherst, MA 01038, USA

Abstract

Persistent programming is concerned with the creation and manipulation of data with arbitrary lifetimes. A requirement of such systems is that the data (including programs) must be capable of evolving and that the evolution should be within the control of the application's programmer. This paper discusses some recent developments in persistent programming that enable controlled evolution. The areas discussed are: the use of type systems, the use of reflection and a new style of programming, only available in persistent object systems, called hyper-programming.

1 Introduction

Persistent programming is concerned with creating and manipulating data in a manner that is independent of its lifetime. The persistence abstraction yields a number of advantages in terms of orthogonal design and programmer productivity [ABC83]. One major advantage is that the abstraction integrates the database view of information with the programming language view [AM85]. For this reason persistent programming languages are sometimes called database programming languages [MA90].

A requirement of any large software system is that the evolution of data and programs should be controllable. Since the uses of data and programs cannot be predicted, it is necessary to support the construction of new software systems which make use of existing data and programs, even when they have been defined independently from current program and data. For large scale, widely used or continuously used systems any alteration to the system should not necessarily require total rebuilding. In these we would expect to support

- the creation and binding of new objects,
- the reuse of existing objects (program and data) by new objects,
- new combinations of existing and/or new objects,
- incremental construction of objects and
- alteration of existing objects (including value and type).

This paper concentrates on mechanisms to support the evolution of data (including program) and the evolution of the schema or type descriptors in persistent object systems. In order to achieve the above goals the following areas are identified:

- type systems,

- reflective systems and
- hyper-programming.

The paper surveys recent work on the above areas and where possible outlines some future research topics.

2 Type Systems

The long term goal of research into type systems is to develop an adequate model of type that meets the computational needs of persistent systems [ADG89]. Ideally we would like a simple set of types, and a type algebra, so that by a succession of operations and the provision of parameters, any data model or conceptual data model can be defined [AM86].

Type systems provide two important facilities within both databases and programming languages, namely data modelling and data protection. Data modelling is performed in databases using data models and in programming languages by a classical type system. In the future the traditional database schema will be regarded as a type. Data protection is provided by integrity constraints in databases, type checking in programming languages and dynamic constraints such as capabilities in operating systems. All these mechanisms require integration into a coherent whole [MBC90].

The issue of type checking is central to this activity. Static checking allows assertions to be made and even proved about a computation before it is executed. It therefore provides a level of safety within the system. Dynamic checking is however sometimes necessary for evolving systems in rebinding or merging of schema. The aim is to pursue the limits of static checking.

There are two approaches currently used to provide static checking for database type systems. The first is constraint specification where constraints over the data for a particular computation are expressed in some language. The checking requires a powerful theorem prover beyond the limits of those currently available. Such systems are usually undecidable and an unsuccessful check may be caused by the limitations of the theorem prover rather than inconsistent constraints. The second approach is to extend classical type systems. These describe decidable types with a simpler syntax which allows the user to better understand type checking failures.

2.1 Extending Type Systems over Databases

Type systems are well understood as mechanisms which impose static safety constraints upon a program. Within a database programming environment, however, the type system takes on a much greater role. Traditionally, data manipulated by a programming language is governed by a type system. Data which persists for longer than the invocation of a program, or data which is shared between programs, is passed from the jurisdiction of the type system. This is necessary due to a common operating system interface which is shared by all applications.

Mechanisms which govern long term data, such as protection and module binding, must be dealt with at the level of this interface. Historically this has the consequence that the type system may not be enforced and knowledge of the typed structure of data may not be taken advantage of.

In a persistent system, the storage of data beyond a single program invocation is handled by programming language mechanisms and no common operating system interface is necessary. The only route by which data may be accessed is through the programming language and so the type system of a single language may be used to enforce protection upon both transient and permanent data. High-level modelling realised using the type system may be relied upon for the entire lifetime of the data, as it never passes outside the language system.

In [MBC90] a number of such mechanisms are presented. In statically checked systems these mechanisms range from the integrity constraints of the language ADABTPL [SFS90], through the statically enforced existentially quantified types of Napier88, encapsulation by scoping, subtype inheritance and the software capabilities of Jones & Liskov [JL78]. In dynamically checked systems, the universal union type **any** of Amber and Napier88 may be used as software capabilities as may dynamically checked database integrity constraints. Hardware enforced capability systems form the most primitive but most efficiently implemented form of the techniques [APW86, RA85].

A major goal in system design is the ability to describe the properties of a system without having to execute it. This ability however conflicts with the flexibility of dynamic checking. At any level of abstraction, the requirement is for static checking that is commensurate with the expressiveness needed for a particular application.

Not all constraints on data may be captured statically. Employing a theorem prover allows more powerful static checking but it may not be possible to prove all the theorems. This could be because they are in error or because the theorem prover is not sophisticated enough computationally to prove them in a reasonable time. A solution is to provide a more dynamic check where this occurs in a system.

The protection of data in database systems is normally achieved by integrity constraints and viewing mechanisms. In a persistent system, however, such mechanisms do not need to be specially provided as they may be programmed within a persistent programming language. Integrity constraints may be programmed within procedural encapsulation and viewing mechanisms may be programmed with a combination of encapsulation and existential data types. The added advantage of existential types over traditional viewing mechanisms is that they are statically type checkable. The referential integrity provided by a persistent store ensures that views occur over the same instance of data, and the copying of data, with its associated integrity problems, is not necessary. The details of these mechanisms are discussed in [CDM90].

One drawback of existentially quantified types, however, is that they are somewhat restrictive within a persistent type system, as values created by an existentially defined package may not be stored and retrieved in different static contexts. This is because values of a witness type are type equivalent only in those cases where type safety is statically provable. This allows existential types to be used without the introduction of any dynamic type checking, but also precludes the kind of dynamic type check associated with the retrieval of a value from the persistent store.

This problem was first discussed in [CM88]. In [OTC90] a new adaptation of these types is explored. The alternative model loses none of the desirable static properties, but is more flexible dynamically and allows a value of a witness type to be stored in one context and retrieved in another. An intuitive explanation of the new model is given along with formal type checking rules.

2.2 Implementation of Persistent Type Systems

Type equivalence checking is well understood within a program, but presents some extra problems in a persistent system which allows the independent compilation of co-operating modules. In a persistent system, data which is shared between modules does not move outside the language's type system and so the type system must address new issues. Commonly, languages which are not persistent can allow such co-operation only by using a store interface which is not type secure, such as a file system.

The traditional database schema is now commonly regarded as a type. This leads to a requirement for the efficient manipulation of types in a persistent system to allow provision of the facilities traditionally found in DBMS for schema editing, use and evolution.

Two models of type equivalence are in common use: name equivalence and structural equivalence [ADG89]. It is shown in [CBC90] that while name equivalence schemes are easier to implement and are more efficient they still have to use structural checks to provide important facilities such as schema merging. On the other hand structural equivalence, generally more flexible and less efficient, can often achieve the same performance as name equivalence.

One important topic of research is how to improve the performance of structural equivalence checking. Much effort has been spent in the investigation of how such checking may be performed. The balance between constructing efficient representations of types in terms of store and the speed of the equivalence algorithm in comparing different representations has been exposed. Two likely contendors for type representations are strings and graphs, which have very different characteristics according to the difficulties in their construction and use. Some preliminary measurements are available and the main conclusion is that where the type schema is large and involves the sharing of types, the graph representation is much more efficient in terms of space. It may however be slower in terms of speed of checking depending on its use within a persistent store [CBC90].

Another aspect of persistent type systems causing concern is the efficient implementation of their related language constructs in a data-intensive applications system. Compilers traditionally use type information for two purposes: the first is to ensure static safety constraints within a program, and the second is to generate more efficient code where this is possible according to the type of the data being manipulated.

Values of different types frequently have different machine-level representations, for reasons of efficiency. The difficulty of implementing polymorphic systems stems from the fact that, as the type may be abstracted over, the representation of a value is not always known statically. This causes problems with all kinds of polymorphism. With parametric polymorphism, operations which may manipulate a value of any type are defined over values whose type, and therefore representation, is not known. With inclusion polymorphism operations such as record dereference may be defined over values whose type is partially abstracted over. Representation-dependent information, such as record offsets, may not always be calculated statically. With ad-hoc polymorphism there is a requirement for operations with different semantic meanings to be executed according to the type of the operands, which may be abstracted over in some systems.

A number of solutions to these problems have been proposed and successfully implemented [Mil83, Lis81, DD79, Mat85, BMS80, Fai82, Tur87]. However the performance issues in a persistent system are significantly different from conventional languages, and all of these solutions appear unnecessarily expensive. Two new mechanisms for the more efficient implementation of polymorphism within persistent systems have been proposed. One of these solves the problems encountered when a fixed number of operations is available on a value whose type is fully abstracted over. This is suitable for the implementation of parametric polymorphism, ad hoc polymorphism, and existential data types [MP88, OTC90]. This mechanism is described in [MDC91], with revisions in [Con90], and has been successfully used in the implementation of the persistent language Napier88.

The other proposal is for a mechanism which allows type-specific operations to be performed on a value whose type is partially abstracted over and may therefore be used to implement inclusion polymorphism. This mechanism is described as a solution to object addressing with multiple inheritance in [CBD89] and its generalisation is described in [Con90].

2.3 Type-safe Evolution and Subtyping

One of the hardest type system problems to be faced in terms of evolution is when an incremental change to a data model is required in the presence of a substantial amount of existing data. There are two possible scenarios for this: when a new attribute of a particular entity is required within the model, or when attributes already modelled are required to be modelled differently. In either case, it is important that all the information previously gathered and stored in the database is preserved.

Of these, only the case of modelling new attributes of existing data has been extensively addressed, for two reasons. Firstly, it is arguably a more common case in the long term evolution of a well designed data-intensive system. Secondly, the well understood mechanism of subtyping appears to give a promising and reasonably tractable partial solution to the problem.

The clean functional semantics of subtyping as proposed by Cardelli [Car84] leaves two further problems in the context of data-intensive systems. The first of these is the provision of an efficient implementation of such a language, as discussed above and addressed in [CDM89]. The second is a complication in the semantics of subtyping caused by the presence of assignment. As assignment is arguably essential for practical purposes in a data-intensive applications system, a solution to the semantic incongruity must be found.

The incompatibility between subtyping in its most general form and languages denoting mutable values was first identified by Albano [Alb83]. In [CMM91] the problem is given a general treatment, including an analysis of the factors which combine to cause it and a spectrum of solutions a language designer may take in order to preserve the soundness of a database programming language type system. The solutions are presented in three categories: limiting the context in which subtyping is allowed, modelling mutability within the type system and allowing for the dynamic failure of substitution operations.

One way of safely including the advantages of subtyping in a language with store semantics is to somehow restrict the generality of its application. One such scheme is under preliminary investigation and first results are published in [CM92]. The idea is based on the well-known mechanism of bounded universal quantification and the type system under consideration consists of the introduction of this mechanism as the only subtyping mechanism. This restriction means that all types inferred or expressed in a program are exact types, except in cases where they are explicitly abstracted over. Some new types, known as open and closed quantifier types, are necessary to allow all denotations to be associated with accurate type descriptions.

It is believed that the proposed system gives maximum desirable expressability within a sound, statically checkable type system. This is at the cost however of a what appears to be a rather complex type system and research continues on the subject.

2.4 Constraint Checking

Constraint checking represents an approach to types based on predicates stating invariants that must hold over changes to persistent objects. The simplest example of this is probably the sub-range types from most programming languages, e.g. ADA's subtypes with range constraints. Such types lead to notoriously difficult type-checking problems, which are normally resolved by dynamic checking. In databases, such protection is the domain of integrity constraint maintenance and can involve both static and dynamic checking.

The basic idea is quite simple: any type can be refined by adding a predicate on values of the type; only values obeying the predicate are in the new subtype. In databases, functional dependencies and referential integrity can be expressed in this way. A functional dependency is a predicate added to a relational type. Referential integrity is captured as a predicate stating that a column of one relation (a "foreign key" column) is contained in a key column of another (or the same) relation. This predicate, like other interrelational constraints, must be added to the database type itself. The problem is that non-trivial theorem proving is now required as part of static type-checking in order to avoid very expensive dynamic checks, and for quite simple predicate and manipulation languages static type-checking is either computationally intractable or undecidable. One response to this difficulty is to limit the constraint and manipulation languages as well as raising their level of abstraction. With limited languages some effective theorems can be used to design procedures for checking computations and generating optimised run-time checks [BB81,BBC80,HI85,MH89,SS89]. Redundant data and special run-time integrity subsystems can be used to speed up checking. Most approaches in the literature work independently of a type system.

It is possible to integrate a theorem prover into the type checker in order to maximize the amount of static type checking achievable in the presence of predicates. The setting in which this appears to be feasible consists of high level languages limited in expressiveness and with the same formal base for the predicate and update languages. It is also possible that static predicative type checking will only be effective with small programs such as typical teleprocessing database transactions.

One benefit of a theorem prover embedded in a type checker is that a broader range of conditions could be used in specifying bounded universal quantification. Normally the conditions on the instantiating types of bounded universally quantified types only specify the existence of operators. This is easy to check. If a theorem prover is available, further predicates on the operators can be added. These conditions need to be proved from the properties of any instantiating type. This can be expensive and even incomplete, though it typically needs to be done in response to a compile-time resolvable declaration, not a run-time action.

Significant effort has gone into building an efficient, though necessarily incomplete, type checker for the set-oriented database programming (or specification) language ADABTPL [SS89]. Efficient proofs of integrity constraint maintenance have been achieved in ADABTPL by limiting the set of constraints and update primitives and by building a set of generic theorems about the interaction of the primitives.

Many of the ADABTPL theorems are higher order theorems, and engineering effective ways to use them is crucial to achieving efficiency. It is quite difficult to characterize the limits of the current ADABTPL techniques. It is, in effect, an expert system with heuristics coded in Lisp and rules that are all proved theorems. It can be improved by adding to its heuristics or to its theorems. The incompleteness of the reasoning is one of the aspects that is most troubling about this approach. Failing to prove that predicates are not maintained is ambiguous in this setting. It means either that the program can produce invalid data or it is too difficult to prove that it obeys the predicates.

There are several responses to a failed proof. The first is to add the unproved residue of the theorem to the transaction as a precondition. In many cases this is the proper response, indicating that the designer forgot the precondition. In other cases, the residue indicates that certain updates were left out. A simple case illustrates this. Suppose a database includes employees and dependents and each dependent is constrained to relate to an employee. If we attempt to prove that a transaction consisting solely of a delete of an employee preserves the constraint, then an unprovable residue will be left at the end of the unsuccessful try. The residue will be the predicate stating that there are no dependents related to the employee being deleted. This predicate could be added as a precondition of the transaction, indicating that the transaction was only intended for use in deleting employees with no dependents. However, it may be the case that the designer forgot to include the deletion of the dependents in the transaction. In this case, the transaction is rewritten to include the deletions and the proof is attempted on the new transaction.

Because of the incompleteness of non-trivial, efficient theorem proving, the residue may in fact be true in all valid databases, but unprovable by the static checker. There are three ways of handling this case open to a designer.

- Allow the check to be made anyway, since the informal proof that convinced the designer that the residue was a theorem may be faulty.
- Simplify the transaction or integrity constraints specification without changing their meaning or change the transaction or constraints if consideration of the precondition reveals that they did not have the intended semantics and retry the proof.
- Tell the system that the check need not be made, thus overriding the checking in this case.

A description of the kind of theorem proving that is needed in assuring the maintenance of predicate-based integrity will clarify some of its limits and costs. The following theorem is the kind that needs to be proved to show that a transaction T leaves a database in a consistent state if it was consistent before the transaction. IC is the single predicate collecting all the different predicates constituting the specification of database integrity. I stands for the input to the transaction. (The input needs to be checked to see that it obeys its type constraints. This is a dynamic check, of course.)

$$\mathrm{IC(DB)} \rightarrow \mathrm{IC(T(DB,I))}$$

A form of theorem that is quite useful in proving such theorems is the following:

$$\mathrm{CONSTR(UPDATE(D))} = \mathrm{SIMPLER(D)}\ \&\ \mathrm{CONSTR(D)}$$

The idea behind this form is that the integrity of an updated database will be expressed in terms of constraint predicates (CONSTR) on updated data elements (D), the lefthand side of the equation. The righthand side form, which will replace expressions like the lefthand side during a proof, is a good form since the latter part (CONSTR(D)) will occur in the antecedent of the theorem, expressing the assumption that the database is initially consistent. This allows CONSTR(D) to be removed immediately from the proof obligation. Thus we are left with SIMPLER(D) where CONSTR(UPDATE(D)) appears in the theorem. An example will make this clear.

A theorem (assuming i is not in R) that is useful in maintaining a key constraint on a relation is:

$$\mathrm{key(insert(i,R),K)} = \mathrm{i.K} \notin \mathrm{project(R,K)}\ \&\ \mathrm{key(R,K)}$$

This states that the condition of a column K being a key of a relation R after a new tuple i has been inserted is equal to the K component of the new tuple not being in the K projection of R and R being keyed on K. This is useful in proving that a particular insert does not violate the key constraint. Since the second term in the righthand side is assumed to be true, the proof of the key property in the updated relation has been reduced to a check for non-membership in a projection. Further proving may verify that the insert only occurs in a position in which the non-membership is assured, in which case this part of the overall theorem has been proven. If the non-membership cannot be proven, it can be added to the transaction as a dynamic check. This case is quite simple but indicates the basic approach to using theorem proving to assure that predicates are invariants of transactions.

There are two further points to be made. The first is that the SIMPLER predicate should be better than the constraint on the updated data in some sense, either by having less computational complexity or in facilitating further reduction, in order for this approach to be effective. The second point is that a uniform formal base and limitations on the integrity constraints and update functions are probably necessary in order to build a set of effective theorems (and heuristics) for this application of theorem proving. This approach embodies both static checking, in the form of the theorem proving, and dynamic checking, in the form of the reduced integrity constraints executed at run-time. Other than the cost of the theorem proving, which in ADABTPL currently runs to less than half a minute for six hundred term theorems on modern workstations, and the cost of the run-time checks, this approach requires building effective theorem bases for different styles of databases, limiting language expressiveness, and formalizing constraint and update languages in a uniform manner.

3 Reflective Systems

Reflection has been used to control the production and evolution of data and programs in database and programming language systems. Reflective systems allow their own structures to be altered from within in two ways: by a program altering its own interpretation or by it changing itself. The first of these, which is common in object-oriented systems, is called behavioural reflection and the second we have named linguistic reflection [SSS92]. Linguistic reflection allows a program to generate code that is integrated into the program's own execution. Such reflection can be used to facilitate both the production and evolution of programs. It is of special interest in database programming languages since in these languages supporting evolution is a major requirement. In current systems it has been used to attain high levels of genericity [SFS90], accommodate changes in systems [DB88, DCK89], implement data models [CAD87, Coo90b], optimise implementations [Coo90a, FS91] and validate specifications [SSF92, FSS92].

The integration of strong typing, compilation systems and reflective expression in a persistent environment is of particular concern. Two techniques for type-safe linguistic reflection have evolved: compile-time linguistic reflection and run-time linguistic reflection. The importance of type-safe linguistic reflection is that it provides a uniform mechanism for production and evolution that exceeds the capabilities of present database programming languages [SSS92].

Linguistic reflection involves defining representations of the syntactic structures of a language within the same language. Compile-time linguistic reflection allows the user to define generator functions which produce representations of program fragments. The generator functions are executed as part of the compilation process. After the generators execute, their results are then viewed as program fragments and become part of the program being compiled.

In run-time linguistic reflection the mechanism is concerned with the construction and binding of new components with existing components in an environment. The technique involves firstly the use of a compiler that can be called dynamically to compile newly generated program fragments and secondly a linking mechanism to bind these new program fragments into the running program.

In order to maintain type-safety each generated program must be type checked in both compile-time and run-time linguistic reflection. Type checking the generators, as opposed to the generated programs, for type correctness of all their generated programs is a second order type checking problem and is, we believe, undecidable in general. This is the subject of future research.

Two examples of the uses of linguistic reflection are:

- abstraction over types and
- accommodating evolution in strongly typed persistent systems.

Abstractions over types are useful when the details of a computation depends significantly on the details of its input types. A generic natural join function provides such an example. Here the details of the input types, particularly the names of the tuple components, significantly affect the algorithm and the output type of the function, determining:

- the result type,
- the algorithm to test whether tuples from both input relations match on the overlapping fields and
- the code to build a relation having tuples with the aggregation of fields from both input relations but with only one copy of the overlapping fields.

Some functions, such as natural join, can not be written in sophisticated typed languages such as ML [Mil83], Quest [Car89] or Napier88 [MBC89] since the algorithm and output type depend on the input types. However, the specification of a generic natural join function may be achieved by compile-time linguistic reflection as long as the types of the input relations are known at compile-time and by run-time reflection otherwise. The details are given in [SSS92].

Another way linguistic reflection may be used is in accommodating the evolution of strongly typed persistent object stores. Characteristics of such stores are that the type system is infinite and that the set of types of existing values in the store evolves independently from any one program. This means that when a program is written or generated some of the values that it may have to manipulate may not yet exist and their types may not yet be known for inclusion in the program text. For strong typing these values must have a most general type but in some applications their specific types can only be found once they have been created.

An example of such a program is a persistent object store browser [DB88, DCK89, KD90] 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 which 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 PS-algol's **pntr** [ABC83], Amber's **dynamic** [Car85] or Napier88's **any** [MBC89]. Once the type of the object is known a program fragment may be generated to browse over it and, by reflection, included in the browser program. Again the details are contained in [SSS92].

Applications of linguistic reflection in the context of database programming languages have stimulated the development of the technology described above. These applications address the following problems:

- attaining high levels of genericity [SFS90],
- accommodating changes in systems [DB88, DCK89],
- implementing data models [CAD87, Coo90b],
- optimising implementations [Coo90a, FS91], and
- validating specifications [SSF92, FSS92].

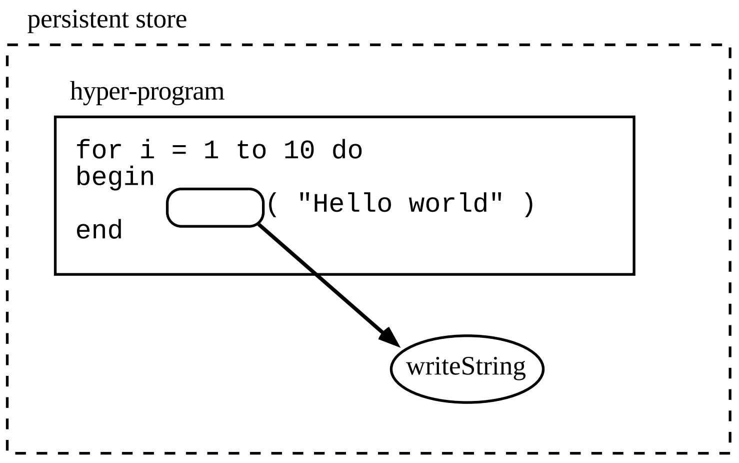
An analysis of the mechanisms and anatomy of type-safe linguistic reflection has been developed in [SSS92] along with a more detailed analysis of the above uses.

Type-safe linguistic reflection has been achieved in PS-algol [PS88], Napier88 [MBC89] and TRPL [She90] by type checking each generated program segment, which is necessary when the complete programming language can be used to write generators. Limiting the language subset available for writing generators may allow the generators to be type checked for the type of all output at one time [Kir92]. This is a topic for future research. Other work to be done includes combining the two styles of reflection presented here, finding well engineered means of writing linguistically reflective code and exploring the relationship of linguistic reflection with other kinds of reflection [Mor92].

4 Hyper-Programming

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

Persistent systems allow the persistent environment to participate in the program construction process. Where the language is persistent, the persistent store can participate in the program construction process. The programmer composes programs interactively by navigating the persistent store and selecting data items to be incorporated into the programs. This requires direct links to the persistent data items to be represented in the program source. By analogy with hyper-text, where a piece of text contains links to other pieces of text, this source representation is called a hyper-program. Figure 2 shows an example of a hyper-program.



```
for i = 1 to 10 do
begin
        ( "Hello world" )
end
```

Figure 2: A hyper-program.

The hyper-program contains both text and a token that denotes a data item in the store, a procedure to write out strings.

Figure 3 shows how the hyper-program might appear to the programmer during editing. A more detailed description of a hyper-programming user interface can be found in [KCC92].

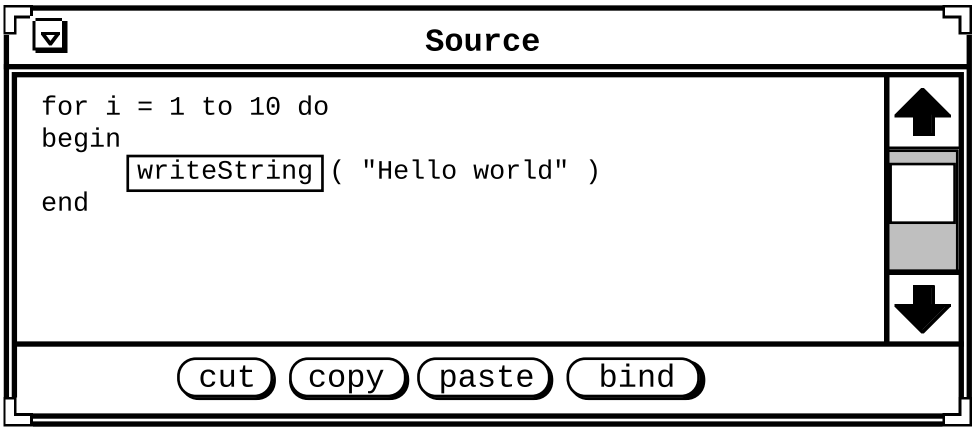

```
for i = 1 to 10 do
begin
      writeString ( "Hello world" )
end
```

Figure 3: A hyper-program editor.

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

The benefits of hyper-programming include:

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

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

Secondly, the hyper-program source representations must be denotable values in the programming language. Linguistic reflective facilities [SSS92] are required to support the conversion of hyper-program representations into executable programs. Where the executable programs produced by reflecting over the hyper-programs are themselves language values, a suitable representation is required. One possibility is to use procedure closures; these are already supported as first class values in a number of languages.

A third requirement is for tools that provide the programmer with the graphical representation of the persistent store. The representation shows the values, locations and types in the store and the links between them. The programmer can point to the representations of specific data items and obtain tokens for them to be incorporated into hyper-programs.

To be useful in practice a hyper-programming system will also have to support additional facilities for ‘programming in the large’, that is, building large applications from smaller components. These include facilities for controlling the sharing of components between applications, for limiting the visibility of some components for protection reasons, and for imposing a degree of partitioning on the persistent store to aid intellectual manageability and execution efficiency. A model to support these facilities, the *hyper-world* model, is described in [KCC92].

5 Conclusions

This paper concentrated on new mechanisms for controlling evolution of data (including program) and the evolution of the schema or type descriptors in persistent object systems. In order to achieve these goals the following areas were identified:

- the use of type systems ;
- the use of reflective systems ; and
- the use of hyper-programming.

The long term goal of research into type systems is to develop an adequate model of type that meets the computational needs of persistent systems [ADG89]. Ideally we would like a simple set of types, and a type algebra, so that by a succession of operations and the provision of parameters, any data model or conceptual data model can be defined [AM86].

The development of type systems to achieve the above includes:

- extending type systems over databases [MBC90, CDM90, OTC 90],
- the implementation of persistent type systems [CBC90, MDC90, Con90],
- type-safe evolution and subtyping [CDM89, CMM91, CM92], and,
- constraint checking [SS89].

Reflection has been used to control the production and evolution of data and programs in database and programming language systems. The applications of reflection include:

- attaining high levels of genericity [SFS90],
- accommodating changes in systems [DB88, DCK89],
- implementing data models [CAD87, Coo90b],
- optimising implementations [Coo90a, FS91], and
- validating specifications [SSF92, FSS92].

The invention of a new style of program construction that is only available in persistent systems, called hyper-programming, is also described and its use in the evolutionary process discussed. The advantages of hyper-programming are:

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

References

[ADG89] Albano, A., Dearle, A., Ghelli, G., Marlin, C., Morrison, R., Orsini, R. & Stemple, D.
"A Framework for Comparing Type Systems for Database Programming Languages". 2nd International Workshop on Database Programming Languages, Oregon (1989), 203-212. In **Database Programming Languages**. (Eds. R.Hull, R.Morrison & D.Stemple). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, 170-178.

[ABC83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An Approach to Persistent Programming." The Computer Journal, 26, 4, (1983), 360-365.

[AM85] Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment". Proc of the Appin Workshop on Data Types and Persistence, Universities of Glasgow and St Andrews, PPRR-16, (August 1985), 1-25. In **Data Types and Persistence** (Eds Atkinson, Buneman & Morrison) Springer-Verlag. (1988), 3-20.

[AM86] Atkinson, M.P. & Morrison, R.
"Integrated Persistent Programming Systems". 19th International Conference on System Sciences, Hawaii, U.S.A., (January 1986), 842-854.

[Alb83] Albano, A.
"Type Hierarchies and Semantic Data Models". ACM SIGPLAN 83, San Francisco, (1983), 178-186.

[APW86] Anderson, M., Pose, R.D. & Wallace, C.S.
"A Password-Capability System", The Computer Journal, 29, 1, (1986), 1-8.

[BB81] Bernstein, P. A. & Blaustein, B. T.
"A Simplification Algorithm for Integrity Assertions and Concrete Views". Proc. of the Fifth International Computer Software and Applications Conference, (1981), 90-99.

[BBC80] Bernstein, P. A., Blaustein, B. T, & Clarke, E. M.
"Fast Maintenance of Semantic Assertions Using Redundant Aggregate Data". Proc. of the Sixth International Conference on Very Large Databases, (1980), 126-136.

[BL84] Burstall, R. & Lampson, B.
"A kernal language for abstract data types and modules". Proc. international symposium on the semantics of data types, Sophia-Antipolis, France (1984).

[BMS80] Burstall, R., McQueen, D. & Sanella, D.
"Hope : An experimental applicative language". ACM Lisp Conference, New York, (1980), 136-143.

[CAD87] Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment". 13th VLDB, Brighton, UK (1987), pp 117-126.

[Car84] Cardelli, L.
"A Semantics of Multiple Inheritance". In **Lecture Notes in Computer Science**, 173, Springer - Verlag, (1984), 51-67.

[Car85] Cardelli, L.
Amber. Tech. Report AT7T. Bell Labs. Murray Hill, U.S.A. (1985).

[Car89] Cardelli, L.
"Typeful Programming". Technical Report 45, Digital Systems Research Centre, California (May 1989)

[CBC90] Connor, R.C.H., Brown, A.L., Cutts, Q.I., Dearle, A., Morrison, R. & Rosenberg, J.
"Type Equivalence Checking in Persistent Object Stores". 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA. (1990), 151-164. In **Implementing Persistent Object Bases : Principles and Practice** (Eds Dearle, Shaw & Zdonik). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, (1990), 154-170.

[CDM89] Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L.
"An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance". Conference on Object Oriented Programming : Systems, Languages and Applications, New Orleans, (October 1989), 279-286.

[CDM90] Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L.
"Existentially Quantified Types as a Database Viewing Mechanism". International Conference on Extending Database Technology, Fondazione Cini, Venice, (March 1990). In **Lecture Notes in Computer Science**. (Eds. F.Bancilhon, C.Thanos & D.Tsichritzis). 416. Springer-Verlag (1990), 301-315.

[CM88] Cardelli, L. and MacQueen, D.
"Persistence and Type Abstraction". Proc of the Appin Workshop on Data Types and Persistence, Universities of Glasgow and St Andrews, PPRR-16, (August 1985). In **Data Types and Persistence** (Eds Atkinson, Buneman & Morrison), Springer-Verlag, Heidelberg, (1988), 31-42.

[CMM91] Connor, R.C.H., McNally, D. & Morrison, R.
"Subtyping and Assignment in Database Programming Languages". 3rd International Workshop on Database Programming Languages, Nafplion, Greece, (August 1991), 305-324. In **Database Programming Languages: Bulk Types & Persistent Data** (eds P.Kanellakis & J.W.Schmidt). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, (1992), 363-382.

[CM92] Connor, R.C.H. & Morrison, R.
"Subtyping without Tears". Australian Computer Science Conference 15, Tasmania (1992), 209-225.

[Con90] Connor, R.C.H.
"Types and Polymorphism in Persistent Programming Systems", Ph. D. Thesis, St Andrews, 1990.

[Coo90a] Cooper, R.L.
"Configurable Data Modelling Systems". Proc. 9th International Conference on the Entity Relationship Approach, Lausanne, Switzerland (1990), 35-52.

[Coo90b] Cooper, R.L.
"On The Utilisation of Persistent Programming Environments". PhD Thesis, University of Glasgow Research Report CSC 90/R12 (1990).

[DB88] Dearle, A. & Brown, A.L.
"Safe Browsing in a Strongly Typed Persistent Environment". The Computer Journal 31, 6, (1988), 540-545.

[DCK89] Dearle, A., Cutts, Q.I. & Kirby, G.N.C.
"Browsing, Grazing and Nibbling Persistent Data Structures". Third International Conference on Persistent Object Systems, Newcastle, Australia (1989), 96-112. In **Persistent Object Systems**. (Eds. J.Rosenberg & D.Koch). Springer-Verlag, 56-72.

[DD79] Demers, A. & Donahue, J.
"Revised report on Russell". Technical report TR79-389, (1979), Cornell University.

[Fai82] Fairbairn, J.
"Ponder and its type system". University of Cambridge. Tech Report 31 (1982).

[FS91] Fegaras, L. & Stemple, D.
"Using Type Transformation in Database System Implementation". Third International Workshop on Database Programming Languages, Nafplion, Greece (1991), 289-305. In **Database Programming Languages: Bulk Types & Persistent Data** (eds P.Kanellakis & J.W.Schmidt). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, (1992), 337-356.

[FSS92] Fegaras, L., Sheard, T. & Stemple, D.
"Uniform Traversal Combinators: Definition, Use and Properties". Eleventh International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York (1992).

[GR83] Goldberg, A. & Robson, D.
Smalltalk-80: The language and its Implementation. Addison Wesley (1983).

[HI85] Hsu, T. & Imielinski, T.
"Integrity Checking for Multiple Updates". Proc. of the ACM-SIGMOD International Conference on Management of Data, (1985), 152-168.

[JL78] Jones, A. K. & Liskov, B.
"A Language Extension for Expressing Constraints on Data Access", CACM, 21, 5, (1978), 358-367.

[KCC92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R.
"Persistent Hyper-Programs". 5th International Conference on Persistent Object Systems, Italy (1992).

[KD90] Kirby, G.N.C. & Dearle, A.
"An Adaptive Graphical Browser for Napier88". University of St Andrews Research Report CS/90/16 (1990).

[Kir92] Kirby, G.N.C.
"Persistent Programming with Strongly Typed Linguistic Reflection". In Proc. 25th International Conference on Systems Sciences, Hawaii (1992), 820-831.

[Lis81] Liskov, B.H.
"CLU Reference Manual". In **Lecture Notes in Computer Science**. Springer-Verlag 114 (1981).

[Mat85] Matthews, D.C.J.
"Poly manual". Technical Report 65 (1985), University of Cambridge, U.K.

[MA90] Morrison, R. & Atkinson, M.P.
"Persistent Languages and Architectures". International Workshop on Computer Architectures to Support Security and Persistence of Information, Universität Bremen, West Germany, (May 1990). (Invited paper). In **Security and Persistence**. (Eds. J.Rosenberg & L.Keedy), Springer-Verlag, 9-28.

[MBC89] Morrison, R., Brown, A.L., Connor, R. & Dearle, A.
"The Napier88 Reference Manual". Universities of Glasgow & St Andrews Persistent Programming Research Report 77-89 (1989)

[MBC90] Morrison, R., Brown, A.L., Connor, R., Cutts, Q.I., Dearle, A., Kirby, G., Rosenberg, J. & Stemple, D.
"Protection in Persistent Object Systems". In **Security and Persistence**. (Eds. J.Rosenberg & L.Keedy), Springer-Verlag (1990), 48-66.

[MBD90] Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"On the Classification of Binding Mechanisms". Information Processing Letters. 34, (Feb 1990), 51-55.

[MDC91] Morrison, R., Dearle, A, Connor, R.C.H. & Brown, A.L.
"An ad hoc Approach to the Implementation of Polymorphism". ACM.TOPLAS. (July 1991), 342-371.

[MH89] McCune, W. & Henschen, L.
"Maintaining State Constraints in Relational Databases". Journal of the ACM 36, 1, (January 1989), 46-68.

[Mil83] Milner, R.
"A Proposal for Standard ML". University of Edinburgh CSR - 157 - 83 (1983).

[Mor92] Morrison, R.
"Reflective Programming : An Application Generator Technology". Australian Database Conference, Melbourne (1992). Invited Talk.

[MP88] Mitchell, J.C. & Plotkin, G.D.
"Abstract Types have Existential type". ACM TOPLAS 10,3 (1988), 470-502.

[OTC90] Ohori, A., Tabkha, I., Connor, R.C.H. & Philbrow, P.
"Persistence and Type Abstraction Revisited", 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA. (1990), 137-150. In **Implementing Persistent Object Bases : Principles and Practice** (Eds Dearle, Shaw & Zdonik). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, (1990), 141-153.

[PS88] "PS-algol Reference Manual".
Universities of St Andrews and Glasgow PPRR39 (1988).

[RA85] Rosenberg, J. & Abramson, D.A.
"A Capability-Based Workstation to Support Software Engineering". Proceedings of 18th Annual Hawaii International Conference on System Sciences, (1985), 222-230.

[SFS90] Stemple, D., Fegaras, L., Sheard, T. & Socorro, A.
"Exceeding the Limits of Polymorphism in Database Programming Languages". In **Lecture Notes in Computer Science**, 416, Springer-Verlag (1990), 269-285.

[She90] Sheard, T.
"A user's Guide to TRPL: A Compile-time Reflective Programming Language". COINS, University of Massachusetts Technical Report 90-109 (1990).

[Str67] Strachey, C.
"Fundamental concepts in programming languages". Oxford University Press, Oxford (1967).

[SS89] Sheard, T. & Stemple, D.
"Automatic Verification of Database Transaction Safety". ACM Transactions on Database Systems 12, 3 (September, 1989), pp. 322-368.

[SSF92] Stemple, D., Sheard, T. & Fegaras, L.
"Linguistic Reflection: A Bridge from Programming to Database Languages". 25th Hawaii International Conference on Systems Sciences (1992), 844-855.

[SSS92] 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". Submitted for publication ACM.TOPLAS.

[Tur87] Turner, D.A.
Miranda System Manual. Research Software Ltd. Canterbury, England. (1987).