This paper should be referenced as:

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. “Exploring the Boundaries of Static Safety in Persistent Application Systems”. In Proc. 18th Australian Computer Science Conference, Adelaide, Australia (1995) pp 99-107.

Exploring the Boundaries of Static Safety in Persistent Application Systems

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

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

{richard, quintin, graham, ron}@dcs.st-andrews.ac.uk

Abstract

Hyper-programming is a technique that allows the programmer to construct persistent code which includes direct references to existing persistent data and code. This mechanism for binding at composition time to persistent objects yields a number of benefits including the ability to perform more static checking earlier in the software life cycle. Early static checking is widely held to improve the safety of programs. Hitherto only limited static checking combined with the dynamic verification of static assertions has been possible in persistent systems such as file systems and object-oriented databases. Through hyper-programming this paper explores the boundaries of static safety in persistent application systems.

1 Introduction

The persistence paradigm tackles the complexity of building long-lived data-intensive applications [1, 2]. Orthogonal persistence means that the length of time for which an item of data persists is independent of the computational model which is used to manipulate it. Programmers are therefore relieved of the burden of writing code to perform data translations between long-term and transient data environments. Persistent application systems manipulate large bodies of long-lived, highly structured and inter-connected data. Traditional databases do not provide sufficiently general computation over this data, whilst traditional programming languages cannot efficiently support the quantity and longevity of the data. The significance of such systems, examples of which are CAD/CAM systems, office automation and CASE tools, is increasing with the general rise in users' aspirations of computer systems.

Translation between data environments typically comprises 30% of the code of data-intensive applications [3]. Its avoidance through persistence not only reduces the overall size of the code body, but also avoids those parts which are most sensitive to changes in the execution environment of the application. Although persistence is a relatively recent concept, industry has been quick to understand the potential benefits; a number of commercial object stores and object-oriented database management systems have been developed, which are being used by a significant and rapidly growing number of applications developers. Examples of these are GemStone [4] and O_2 [5], which have at their core a persistent object store and are examples of persistent systems within a particular programming paradigm.

Statically checked type systems are one established aid to overcoming the economic and social consequences of unreliable software systems. However all currently available commercial persistent systems are weakly or dynamically typed, giving cause for concern in their increasing use.

Early experiments in persistence relied upon dynamic typing [2, 6], but more recent research has resulted in an understanding of how static typing may be combined with persistent applications development. This paper presents some techniques which may be used to achieve this static checking. The motivation throughout is improved reliability of persistent applications through increasingly sophisticated static checking.

2 Persistent type systems

The most significant difference between persistent and non-persistent programming languages is that, in a persistent language, the long-term data is typed. This results in a major shift in the emphasis of type system protection, from one of a safety mechanism over programs to that of a safety mechanism over the entire software system, including both programs and data. Type systems are normally viewed as providing two aids to the programmer: a modelling framework to aid the task of data abstraction, and a protection mechanism which prevents this modelling framework from being improperly used by a program. The difference in persistent systems is that the integrity of the data modelling is enforced over data which outlive the program invocation in which they are created, and are shareable by other program invocations.

The first problem to be addressed in the design of a persistent type system is therefore how to model type system protection over data which escapes from or originates outside the context of a program's text. In a non-persistent language the typechecker is usually invoked during the compilation of each program to

check the consistent use of data modelling. Any data accessed externally, for example from a file or database system, is explicitly converted into the type system framework. In a persistent system however the program text may contain expressions which externally access values in persistent storage. Before any program statement which uses such data is executed, a check must have been made that it will not violate any type system constraints placed on the data at the time it was created.

If type system constraints are checked dynamically there is no extra typechecking problem. When values are created their type system attributes are associated with them in such a way that they may be dynamically accessed whenever a check is required. To add persistence to a language with such a type system is relatively easy; the requirements are for an object store implementation which allows values to be stored between program invocations, and a language mechanism which describes a persistent naming protocol so that persistent values may be denoted. The object formats and typechecking mechanisms need not be altered at all.

Dynamically checked type systems however lead to unreliable code as type errors are detected during execution. In static type systems type errors may always be detected before the execution of a program has commenced. Static type systems are well known to be achievable in non-persistent languages; the challenge is to achieve static typechecking within a persistent system. This seems at first to be an intractable challenge, as persistent programs require the ability to access typed data which is external to the program context. Recent research however has shown how purely static typechecking may be achieved, within a fully integrated persistent programming environment.

3 Persistent Bindings

The time at which bindings are created between programs and persistent data is the earliest point at which type checking can occur. The issue of binding in persistent systems introduces a fundamental dichotomy: on the one hand loose binding is required to allow the incremental evolution of program and data within a system [7]; on the other, tight binding is required to maximise the safety and reliability of individual sub-applications. The requirement for loose binding has resulted in dynamically type checked systems; however the two apparently opposing requirements may be provided in a single system through the following steps:

- Firstly, the introduction of appropriate language constructs allows the points of dynamic binding and type checking to be clearly identified within programs. Programs may be statically bound and type checked with respect to a finite number of static assertions about their execution environment.
- Having provided this, the use of first-class executable forms such as procedures allows meta-programs (application constructors) to perform the required dynamic binding. The meta-programs, when executed, can produce as their result statically bound representations of the applications and place them into the same environment as the data [8].
- Lastly, by providing software construction tools within the persistent environment, direct static bindings may be placed in program source, by executing the required binding during the construction process.

Using the above approach the persistent application space may be constructed from the loose bindings required for the incremental evolution of the overall system, but sub-applications within the system may be constructed in such a way that the required dynamic binding is performed before their execution commences.

4 Towards the Static Typing of Persistent Applications

A persistent programming system must include a dynamic binding mechanism, and in a strongly typed system dynamic binding is necessarily associated with dynamic typing. Importantly however the presence of some dynamic typing in a program does not preclude the static type checking of the rest of the program. The separation of these concepts is the first step towards understanding how persistent applications may be statically type checked.

In [9] it is shown how a dynamic type may be added to a static type system without weakening the static knowledge of types. The Napier88 system [10] contains two dynamically checked types, called *any* and *env*. Type *any* is the infinite union of all types into which values may be injected with their specific type and from which values may be projected onto their specific types. Type *env* is also an infinite union but this time of bindings. It may be used as a persistent store constructor.

The use of values of type *env* in Napier88 effectively allows the description of an access path within the persistent store, combined with an assertion about the type of the value to be found there, to be made in a single statement. Figure 1 shows an example Napier88 program which, if successfully executed, will bind to some complex data value in the persistent store.

```
type exampleType is ... some complex type ...

use PS() with myData : env in
use myData with thisData : exampleType in
begin
        ... code which manipulates thisData ...
end
```

Figure 1 : Napier88 Environment Binding

The program first introduces a description of the type *exampleType*. This description may be an arbitrarily fine-grained description of a data model, using various type constructors such as record, variant and procedure, along with the use of type abstraction mechanisms such as parametric polymorphism and existential quantification. The important point is that if the description does not contain either the *any* or *env* type constructors then the use of any denotation associated with this type may be fully statically type checked with respect to this description.

The only dynamic type checking in the example occurs in the two *use* clauses. The expressions immediately after the occurrences of the keyword *use* are statically verified to be of type *env*; in the program these environments are asserted to contain bindings with the names *myData* and *thisData*, respectively of the types *env* and *exampleType*. These assertions are dynamically tested as part of the execution of the *use* statements; importantly however the rest of the program is statically type checked with respect to the correctness of these assertions. This means that once the main block of the program has been reached then the program is guaranteed to execute without the occurrence of a dynamic type error.

Figure 2 depicts how programs may be bound to the persistent store with largely static type checking. The graph of values inside the store may be described by purely static type definitions; the access points to this graph in the shaded area are the points of dynamic checking, about which assertions are made in programs which use the persistent data.

The collective information in the access points may be regarded as the persistent store schema. The majority of programs which use persistent data are written with respect to an unchanging partial schema description. Whenever this is the case, the set of all required dynamic bindings may be organised as a prelude to the program's execution. If the execution of the binding prelude succeeds, then the execution from that point on cannot fail with a dynamic type error, and the application it represents can therefore be regarded as statically typed.

It may however be impossible to execute such programs if interim changes have been made to the structure of the persistent store. A still safer model may be obtained by moving the applications development process into the same environment as the persistent data.

5 Code Within the Persistent Environment

Figure 2 shows the dynamic binding of a program to the persistent store. If the binding fails, the program cannot be executed. Assuming that the programmer has accurate knowledge about the structure of the persistent store at the time of writing, the dynamic binding may be expected to succeed when the program is first used. Over a period of time, changes to the structure and organisation of the store can be expected to render the program ineffective, even if the data it relies upon are still present and useful within the store. To avoid this problem the concept of a navigation free program is introduced; that is a program whose semantics is independent of the potentially changing navigation route required to initially access the data.

To avoid rebinding the program on every execution it may be represented as a procedure value which is kept within the persistent store itself. This allows the program to contain bindings, which do not require

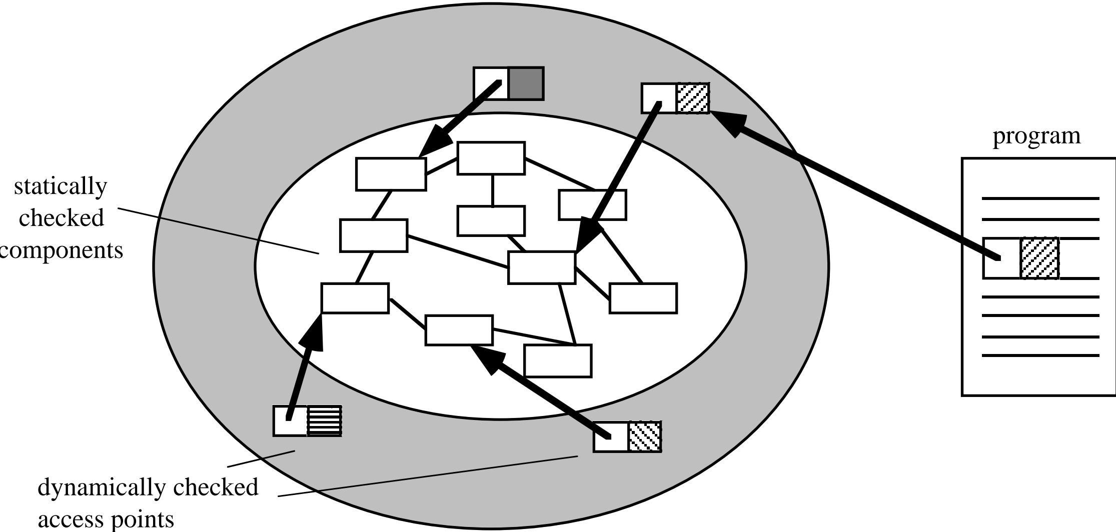

Figure 2 : Code Outside the Persistent Store

to be dynamically resolved, to the data it uses. The technique of interest is that the procedure may be created within another program which first performs the dynamic binding, making this binding static with respect to the procedure declaration. For example, the behaviour of the program in Figure 1 may be encapsulated within a procedure within the persistent store by the program in Figure 3.

```
type exampleType is ... some complex type ...

use PS() with myData : env in
use myData with thisData : exampleType in
begin
        let appl = proc()
        begin
                ... code which manipulates thisData ...
        end

        in PS() let theApplication = appl
end
```

Figure 3 : Placing Code in the Persistent Store

Two lines of code have been added to the original program. The first has the effect of causing the original *begin ... end* block, which contains the code of the application, to become the body of a new procedure *appl*. The last line then creates a new dynamic binding within the persistent store which references this procedure. All references within *theApplication* to the value *thisData* are statically bound and type checked. To execute the program now requires the execution of the program in Figure 4, which will execute correctly as long as the persistent store contains the application with this name; this is equivalent to a normal operating system command line.

```
use PS() with theApplication : proc() in
        theApplication()
```

Figure 4 : Executing the Application

The above binding mechanism is depicted in Figure 5 which shows the code within the persistent store using dynamic bindings in the shaded area. The execution of the code will produce a procedure in which there are no bindings in the shaded area.
The aspiration of achieving a purely statically type checked persistent application has been demonstrated. However by moving the entire software process into the persistent environment further benefit may be gained.

6 An Integrated Programming Environment

The essential concept demonstrated so far is how a persistent store may be used not simply as a repository for strongly and largely statically typed data, but as a base technology in which programs may be coherently stored in the same environment as the data over which they operate. It is now shown how this concept may be further extended so that the persistent environment may be used to encompass the whole software development process [11, 12].

The technology required to bootstrap such a system consists of the usual programming support requirements written as programs which are themselves resident within the persistent store. Such support includes for instance persistent store browsers, text and program editors, window managers and associated graphical toolboxes, and a family of compilers.

In an integrated persistent environment, programs may be constructed and stored in the same environment as that in which they are executed. This means that objects accessed by a program may already be

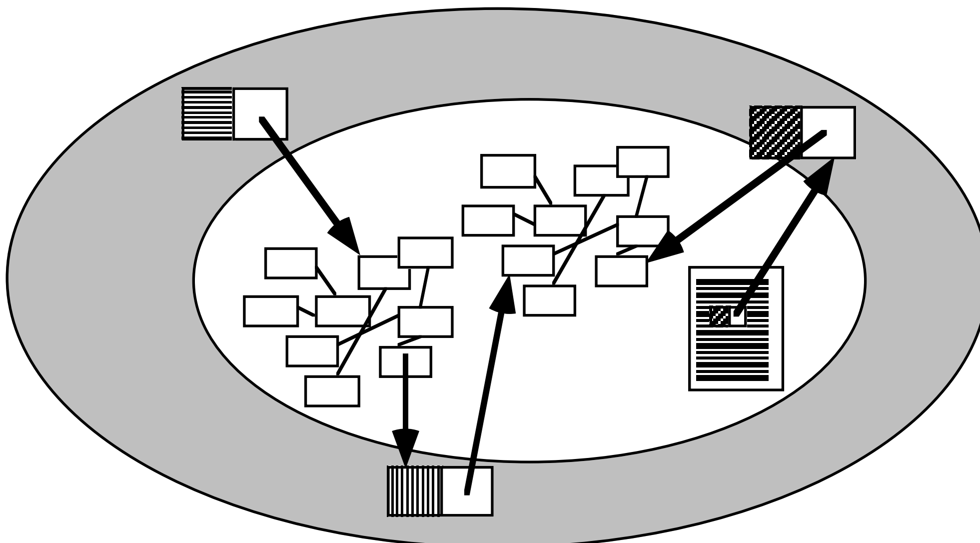

Figure 5 : Code Within the Persistent Store

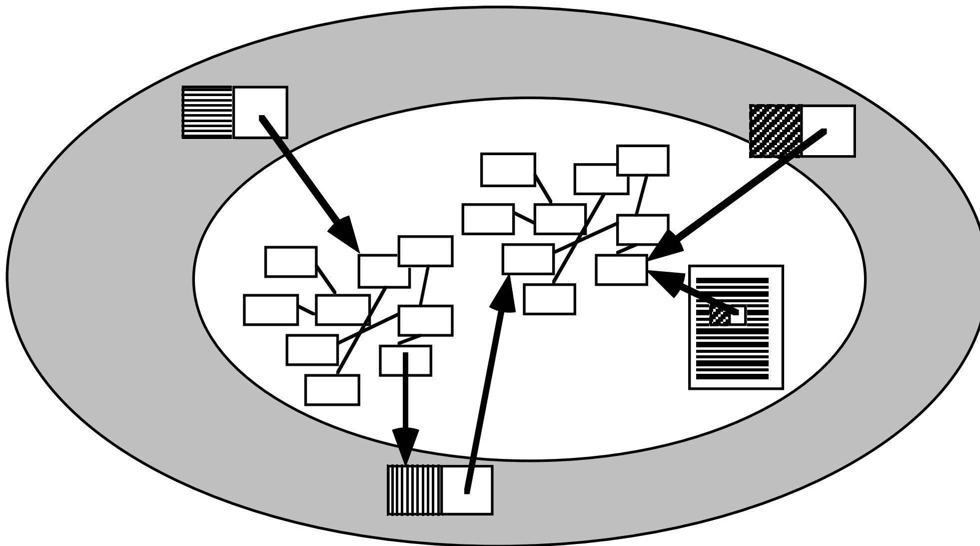

Figure 6 : Hyper-Program Bindings

available when the program is composed. In such cases links to the objects can be included in the program instead of textual descriptions. By analogy with hyper-text, a source program containing both text and links to objects is called a hyper-program [13].

Figure 6 shows an environment with direct links from the source code to the persistent data. Notice that some dynamic bindings remain for flexibility but that unnecessary ones are replaced by static bindings.

Figure 7 shows an example of a hyper-program which contains links to persistent data. The first link is to a procedure value which when called writes a prompt to the user. The program then calls another procedure to read in a name, and then finds an address corresponding to the name. This is done by calling a lookup procedure which is one of the components of a table package linked into the hyper-program. The address is then written out. Note that code objects (*readString* and *writeString*) are treated in exactly the same way as data objects (the table).

More than one program may contain links to a particular component, and the graph of program components can become highly interconnected. The full benefits of hyper-programming are discussed in [13]. They include:

- being able to perform program checking early — access path checking and type checking for linked components may be performed at program construction;
- support for source representations of all procedure closures — free variables in closures may be represented as links, thus hyper-programs may be used for both source and run-time representations of programs;
- being able to enforce associations from executable programs to source programs — links between source and compiled

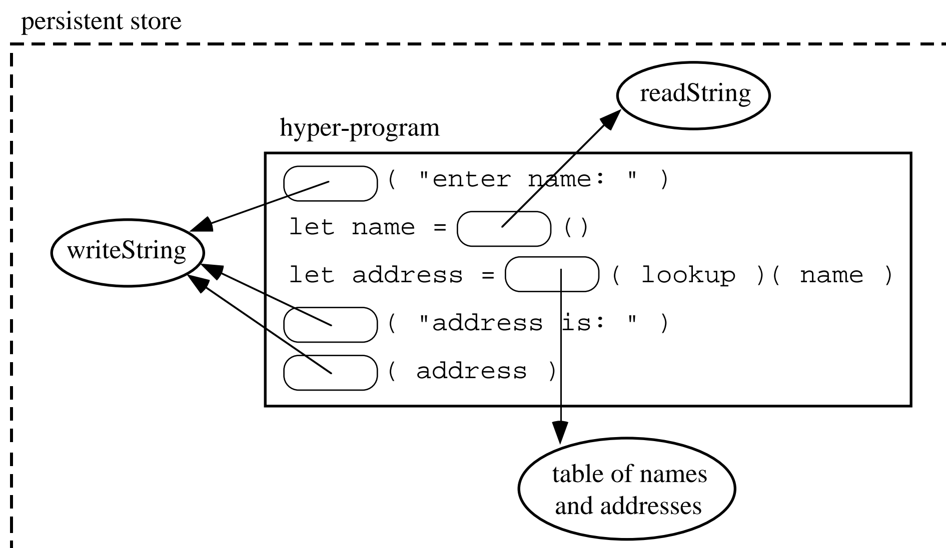

Figure 7 : A Hyper-Program

versions may be used;

- increased program succinctness — access path information, specifying how a component is located in the environment, may be elided; and
- increased ease of program composition — links may be inserted by programmer gesture as well as by typing.

The concept of embedded links within program source is not new, and was first identified in the Flex project at RSRE (DRA) [14]. The major differences in Flex are that there were separate conceptual storage areas for data and programs, and restrictions about circular references in the underlying storage technology.

Figure 8 shows a view of the integrated system during the construction of the same program used in the previous examples. The tools being used here are a hyper-programming editor and a persistent store browser. The browser is used to traverse the persistent space, and includes a mechanism which allows a particular persistent value to be highlighted. In this case it has been used to navigate from the root of persistence to find the value *thisData* as used in the previous examples.

The hyper-programming editor exists in the same environment as the browser. This allows values, once identified by the browser, to be inserted directly into programs. The binding is represented in the program source as a light button itself; pressing on this button invokes the browser over the value.

Notice that once this program is compiled its executable form has the same semantics as the example in Figure 5, in which the executable version of the program is placed in the persistent store environment. In both cases the required dynamic binding has been performed at some point before the construction of the executable form of the program, and the executable form is statically bound and type checked. However the presence of the direct binding in the program source gives major advantages in terms of the ease of program construction, and the succinctness of the source form.

The hyper-programming environment extends the scope of static checking. The presence of hyper-links has the effect of increasing the knowledge which can be gleaned statically about the way the executable code and data are used in conjunction with each other. This extra knowledge may be divided into two categories:

- extra information about the mapping from program source to the data it accesses when it is executed, and
- by keeping bidirectional links, extra information may also be discovered about the mapping from data to those programs which may access them

A example of each of these is now given.

7 Dependent Types

To introduce the usefulness of dependent types a motivating example of dependent bulk types is given.

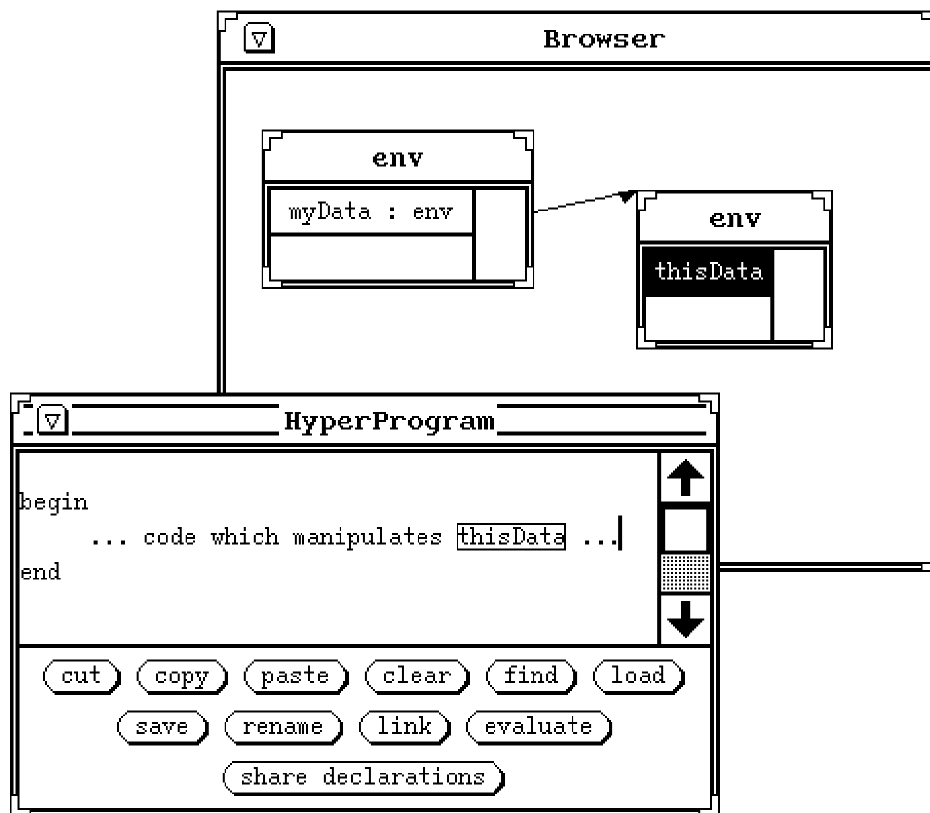

Figure 8 : A Hyper-Programming Session

The semantics of many data bulk type models depends on user-defined attributes such as definitions of element equality, ordering, and other domain predicates. While these attributes are an intrinsic part of the data model, they are not normally treated as part of the type description. This may lead to the occurrence of data modelling errors such as a union operator accidentally being applied to two sets which have different semantics for the equality of their elements.

Directories indexed by Scottish names is an example where the inclusion of element attributes in the type may be a requirement. People may take a different view as to whether the names “MacFarlane”, “Macfarlane”, and “McFarlane” are really different names, but their owners are usually protective of their different forms. If they are used to retrieve data however it is likely that the retriever will not wish to distinguish between them. There is therefore a requirement for different types of dictionary dependent upon the domain equality semantics.

Attributes such as the definition of element equality are typically held with instances of values, rather than with their type [15]. When a function such as union is applied, an arbitrary choice is made between the two possible descriptions of equality. The chosen equality is used not only as an attribute of the newly constructed bulk value, but also to implement the application of the union function itself, as this depends upon the definition of element equality. Thus the semantics of union and similar set functions is compromised as, among other reasons, commutability is lost.

By including these attributes as part of the type, however, it should be possible to define function types over bulk values such that the functions they describe are limited in application to bulk values where the element domain attributes are the same. Any attempt to apply these functions to values with different attributes will be detected by the type system and the execution will be disallowed. A dynamically typed system will generate a type error or exception when the operator is erroneously applied, and a statically typed system will detect such errors before the program which contains them starts to execute.

For dependent types to be structurally equivalent it is necessary for the values upon which they depend to be equal[1]. For the equivalence of dependent types to be statically decidable it is therefore necessary for the values upon which the types depend to be statically available to the typechecker. For some simple examples this is not a problem. For example, array types in Pascal are dependent upon their bounds; as these values are constrained to being manifest constants, the values are available to the typechecker and their equality is statically decidable.

The essential requirement is that any values upon which a type depends are evaluated before any equivalence testing is performed upon the type. This restriction of the general type description can be enforced in a hyper-programming system by restricting the dependencies to being hyper-links.

A full description of such a language mechanism, including the explanation of the necessary restrictions and an introduction to the subject of polymorphism over dependent types, is given in [16]. A major research programme is to generalise the model given here to allow more general user-defined dependent types, along with an investigation of their use.

8 The Integrated Evolution of Program and Data

Collections of data become obsolete when they can no longer meet the changing needs of the applications that they support. Evolution is inevitable in a long running system as the people who use the data, the data itself and the uses to which the data is put all change. This is reflected within a database system by changes to the data, the programs which use the data and the meta-data. Changes to program and data with the invariant of fixed meta-data are normally handled by updates to program libraries and the database 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.

Traditional database technology may be extended by taking advantage of the facilities of an integrated persistent programming environment. 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.

In an integrated persistent environment, programs may be constructed and stored in the same environment as that in which they are executed. For the interest in schema evolution, there is a requirement to locate and translate affected queries and data; the essential elements are at hand in the integrated environment. 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-code source and therefore source code and data translation is possible.

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

1. Locate, from the schema, all affected programs and data.
2. For each program which may be affected, obtain its hyper-code.
3. Locate the points in the hyper-code which access the changed part of the schema and

[1] Clearly there is a general problem when these values are functions. This is however practically soluble for the restricted purposes of structural typechecking, and is described in [16]

edit the hyper-code 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.

Figure 9 shows an example of the interface proposed for the schema editing and change management tool. It is important to note that the concepts described here are not a matter of user interface design; the essential properties of the feasible implementation of this system rely upon the hyper-programming technology, which is in turn dependent upon the integrated 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-code 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.

9 Conclusions

This work is motivated by a belief that programming language systems can provide better support for software engineering than they do at present. Where the language is persistent, the persistent store can partici-

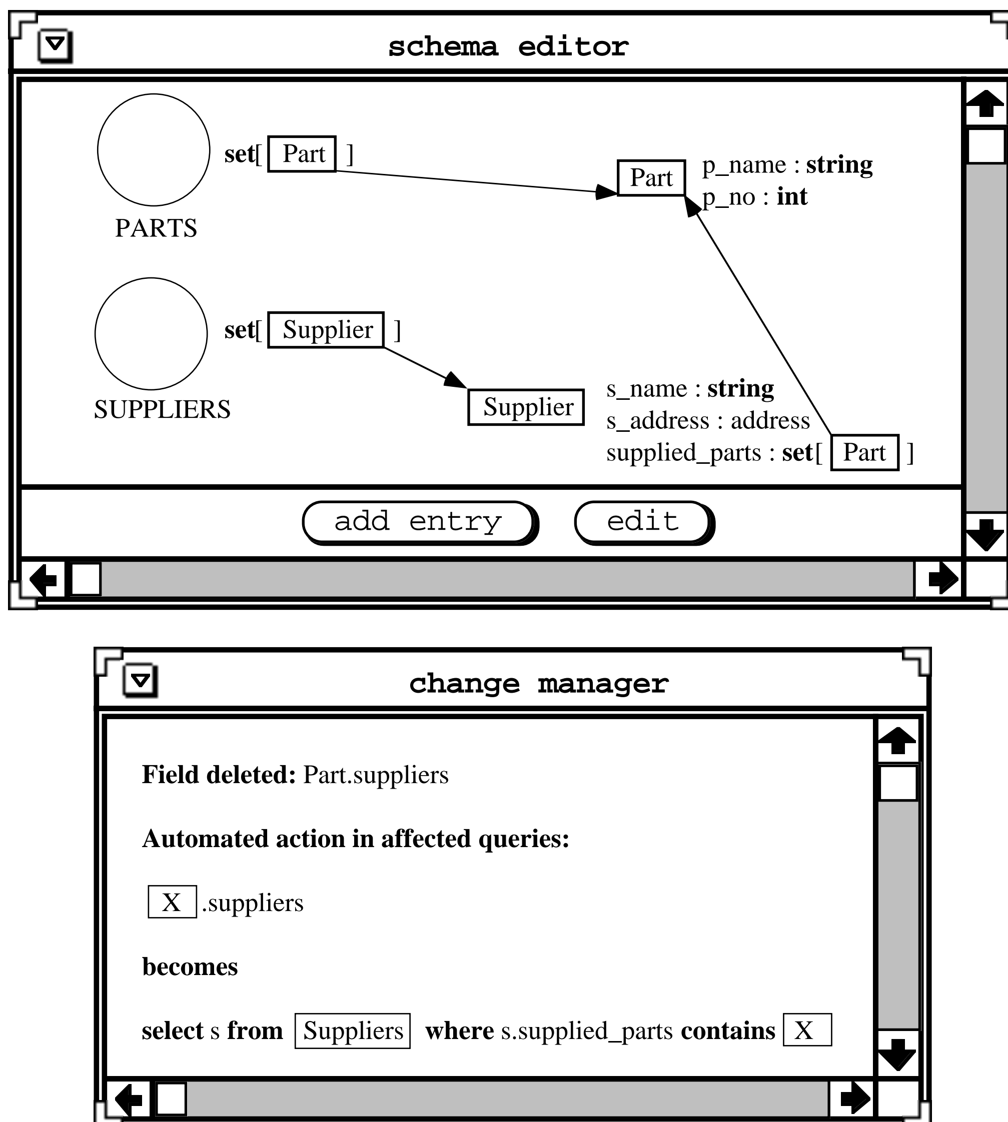

Figure 9 : Schema evolution and change management tool interface

pate 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 hypertext, where a piece of text contains links to other pieces of text, this source representation is called a hyper-program.

The use of hyper-programming in the construction of persistent applications achieves two goals. Firstly the direct binding of the source program to the persistent objects ensures that the program will never fail due to the environment changing between compilation and execution. Secondly static type checking of the persistent values can now take place leading to safer and more reliable persistent application systems.

Building applications within the technology of a fully integrated persistent environment is postulated to make possible new classes of static program analysis which will lead to more reliable software systems. One of the major strengths of the research into such an environment is that they need not be confined to research vehicles such as the Napier88 hyper-programming system, but the technology may be transferred to more commonly used languages and systems such as the current plethora of commercial object stores running under languages such as C^{++} and Smalltalk.

10 References

[1] M.P. Atkinson. Programming Languages and Databases. In *Proc. 4th IEEE International Conference on Very Large Databases*, pages 408-419, 1978.

[2]* M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, Volume 26, Number 4, pages 360-365, 1983.

[3] M.P. Atkinson, R. Morrison and G.D. Pratten. Designing a Persistent Information Space Architecture. In *Proc. 10th IFIP World Congress*, pages 115-120, Dublin, 1986.

[4] B. Bretl, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, M. Williams and D. Maier. The GemStone Data Management System. In *W. Kim and F. Lochovsky (ed) Object-Oriented Concepts, Applications, and Databases*. Morgan-Kaufman, 1989.

[5] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard and F. Valez. The Design and Implementation of O_2, an Object-Oriented Database System. In *K.R. Dittrich (ed) Lecture Notes in Computer Science 334*. Springer-Verlag, 1988, pp 1-22.

[6] PS-algol Abstract Machine Manual. Universities of Glasgow and St Andrews Report PPRR-11-85, 1985.

[7] R. Morrison, M.P. Atkinson and A. Dearle. Flexible Incremental Bindings in a Persistent Object Store. Universities of Glasgow and St Andrews Report PPRR-38-87, 1987.

[8] A. Dearle, Q.I. Cutts and R.C.H. Connor. An Application Architecture Using Type-Safe Incremental Linking. *Journal of Microprocessors and Microprogramming*, Volume 17, Number 3, pages 1993.

[9] M. Abadi, L. Cardelli, B.C. Pierce and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, Volume 13, Number 2, pages 237-268, 1991.

[10]* R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby and D.S. Munro. The Napier88 Reference Manual (Release 2.0). University of St Andrews Report CS/94/8, 1994.

[11] R. Morrison, C. Baker, R.C.H. Connor, Q.I. Cutts and G.N.C. Kirby. Approaching Integration in Software Environments. University of St Andrews Report CS/93/10, 1993.

[12] R. Morrison, C. Baker, R.C.H. Connor, Q.I. Cutts, V.S. Dunstan and G.N.C. Kirby. Exploiting Persistent Linkage in Software Engineering Environments. To Appear: *Computer Journal*, 1995.

[13]* G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, A. Dearle, A.M. Farkas and R. Morrison. Persistent Hyper-Programs. In *A. Albano and R. Morrison (ed) Persistent Object Systems*, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy. Springer-Verlag, 1992, pp 86-106.

[14] M. Stanley. An Evaluation of the Flex PSE. Defence Research Agency, Malvern, England Report 86003, 1986.

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

[16]* R.C.H. Connor, M.P. Atkinson, S. Berman, Q.I. Cutts, G.N.C. Kirby and R. Morrison. The Joy of Sets. In *C. Beeri, A. Ohori and D.E. Shasha (ed) Database Programming Languages*, Proc. 4th International Conference on Database Programming Languages, New York City. Springer-Verlag, 1993, pp 417-433.

*Available:

by *ftp* from *ftp://ftp-fide.dcs.st-andrews.ac.uk/*
pub/persistence.papers or
by *http* from *http://www-fide.dcs.st-andrews.ac.uk/*
Publications.html