On the Unification of Persistent Programming and the World-Wide Web

Richard Connor, Keith Sibson and Paolo Manghi

Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
{richard, sibsonk, manghi}@dcs.gla.ac.uk

Abstract. In its infancy, the World-Wide Web consisted of a web of largely static hypertext documents. As time progresses it is evolving into a domain which supports almost arbitrary networked computations. Central to its successful operation is the agreement of the HTML and http standards, which provide inter-node communication via the medium of streamed files. Our hypothesis is that, as application sophistication increases, this file-based interface will present the same limitations to programmers as the use of traditional file and database system interfaces within programming languages. Persistent programming systems were designed to overcome these problems in traditional domains; our investigation is to reapply the resulting research to the new domain of the Web. The result of this should be the ability to pass typed data layered on top of the existing standards, in a manner that is fully integrated with them. A typed object protocol integrated with existing standards would allow the Web to be used to host a global persistent address space, thus making it a potential data repository for a generation of database programming languages.

1. Overview

There are two fundamentally different approaches to the integration of databases and the World-Wide Web. The first is the "embedded database" approach, where well-behaved and regular segments of data available from the Web are implemented using various forms of database technology. These interfaces may be as simple as accepting embedded SQL queries, or may be more subtle so that the purpose of the database is only to provide efficient and regular storage. In this case the existence of the database metastructure can not be detected from its interface on the Web.

The alternative approach is to view the entire Web as a database. At the current state of the art, the data collection that is the Web has only one thing in common with a traditional database - it is large. To talk of the Web as a single vast database is at best premature: databases have other attributes, which are in fact more significant than their size, such as an enforced semantic model and guarantees about the quality and consistency of the data contained in them. The Web has no such guarantees, and never can in any general sense.

Nonetheless, this is the approach taken in this paper. We seek to improve the potential for the Web to be viewed as a single data collection, significant subsets of which can be treated as global databases. The approach is one of a database programming language where the Web namespace of URLs is overloaded as a naming scheme for typed data which can be manipulated by a strongly and largely statically typed programming language. This language can therefore act as a query language over those parts of the Web whose data is well-behaved, as well as providing a system in which the production of such data can be made easier for a programmer.

The approach proposed is to extend the semantics of a database programming language (more accurately, a higher-order persistent programming language) to include the Web within its semantic domain. One of the most notable features of such languages is that they include passive data and program fragments, in the form of functions and procedures, in the same persistent data domain [AM95]. Therefore the notion of Web data within this context includes higher-order executable code. The investigation described is an attempt to integrate such a system with the existing state of the Web; in particular, there is no suggestion of attempting to replace the Web, by which term we include its host of evolving and emerging standards, with new technology.

2. Introduction

The world-wide web is starting to see an increase in the sophistication of its component documents and applications. Although the origin of the Web was relatively humble browsable text documents, the use of dynamic applications using the same interfaces is becoming commonplace. A number of different models have been developed to support this within the original protocols, including client-side computation via scripts and applets, and server-side computation via CGI and API plugins.

Although these mechanisms allow arbitrary computations to be described within a networked application, the medium by which data is transferred among them is the text file, as defined by the http protocol. This is scarcely a limitation for the majority of current applications, as the common model of computation is based on a client-server interaction whereby the end result is to produce an HTML document to be viewed by a browser. It is envisaged however that, as sophistication of use increases, the requirement for application components to pass more complex data among themselves will increase, and in this context the text file will become a limitation.

The scenario of applications sharing typed data via the medium of text files is exactly that which prompted research into persistent programming systems, and the main intention of the work described here is to reapply that research in the domain of the Web. This paper describes some initial ideas, currently under investigation in the Hippo project at the University of Glasgow [Hip98], which allow the layering of a typed object protocol on top of the HTML and http standards.

One further motivation, of a more pragmatic nature, is that the successful unification of a persistent programming language with the Web will inevitably have

the secondary effect of creating an open persistent system. One of the most interesting fundamental properties of the Web is the autonomy of its users, and any system which attempted to control this would not therefore meet the selected challenge. A major fundamental drawback of orthogonally persistent systems is that they are closed-world, this aspect making them seem unacceptable for much commercial use. Integrating a persistent system with the Web requires the management of an essential set of compromises between a pure model of orthogonal persistence and something that can be engineered within the shared semantic space of the Web. Thus success in the investigation will also address the more general problem of openness in persistent systems; however, it should be admitted, a fundamental incompatibility is not always the best starting point for an investigation!

It must finally be stressed that this investigation is not an attempt to reinvent the industrial standards of CORBA and DCOM. While these successfully allow the modeling of objects with internet interfaces, the main advantages of orthogonal persistence stem from the seamless semantic model of data that is shared across program invocations within a single language. The CORBA and DCOM models have quite a different agenda, in that they provide language-independent interfaces, allowing the objects provided to be implemented by and incorporated into programs in arbitrary languages. The use of CORBA or DCOM objects requires translation from an interface definition written in an Interface Description Language (IDL) into the particular language being used. A major consequence of this is that the descriptive power of the IDL is necessarily compromised to maximise the set of languages that can use the technology, as it must provide something akin to a common subset of their type systems. An IDL can therefore provide at best a way of describing structured scalar types, and references to other interfaces in the same standard. In particular, an IDL could never hope to support types representing values such as function closures or abstract data types. Thus, although there is some overlap, our investigation is into a quite different paradigm of typed internet data.

4. Orthogonal persistence

3.1. Persistence for Data-intensive applications

The concept of orthogonal persistence was identified by Atkinson [Atk78] whilst working on systems where general purpose programming was required over data stored in databases and file systems. The key observation is that data stored in these domains is available externally only in a flat file format, which causes a serious mismatch when it must be translated into a programming language type system. An independent internal investigation by IBM revealed that in some application areas up to 30% of source code was involved purely in translation between the two domains. Furthermore this code attracts high management costs, as it must be changed in detail depending upon the external environment in which it is executed.

Considerable research has been devoted to the investigation of persistence and its application to the integration of database systems and programming languages [ABC+83, AM95]. A number of persistent systems have been developed including

PS-algol [PS88], Napier88 [MBC+89], Galileo [ACO85], TI Persistent Memory System [Tha86] and Trellis/Owl [SCW85]. Most of the object-oriented database products now commercially available provide some form of persistent language interface, and most recently Sun Microsystems and others are involved in a number of experiments in adding the concepts of persistence to Java [AJDS97, GN98]. Whilst many of the systems listed may not be perfect examples of an academic definition of persistence, they clearly borrow the essential concepts, albeit in a commercially viable setting.

The benefits of orthogonal persistence have been described extensively in the literature [AM95], and can be summarised as

- improving programming productivity from simpler semantics;
- removing ad hoc arrangements for data translation and long term data storage; and
- providing protection mechanisms over the whole environment.

Figure 1 gives a simplified pictorial view of the traditional data-sharing scenario. In this diagram, the rectangles represent program invocations, the circles inside them being the executing code and the graphs representing the data manipulated by these invocations.

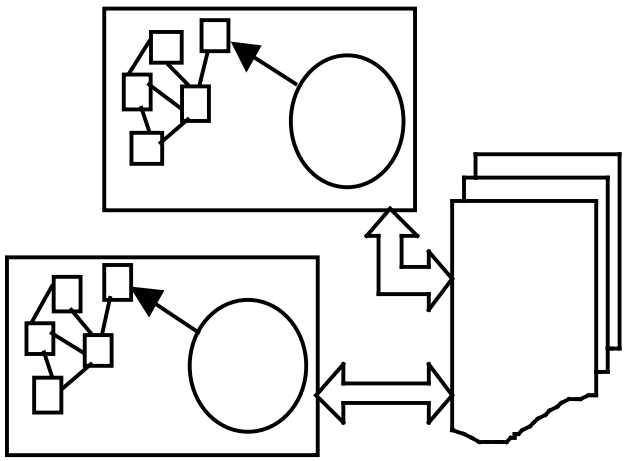

Fig. 1. Traditional file-sharing model of long-lived data

For data to be shared between invocations, either concurrently or after an elapsed time, graphs of complex objects must be explicitly flattened into some format supported by the file system. These objects must then be recreated in the domain of the other invocation, which executes in its own closed semantic domain. In simple terms this increases the burden on the application programmer; however in a sophisticated application domain there are greater problems also, such as the creation of copies of the data and the inability to share data represented by abstract data types. Whilst these problems can be overcome through coding in any computationally complete system, the key observation is that the complexity introduced by the

traditional domain is unnecessary, in that the code which solves these problems is superfluous to the description of the problem the programmer is addressing.

Figure 2 shows the same scenario with a persistent application domain. The rectangle in this figure represents the persistent application environment which, unlike traditional program language environments, encompasses all the data within the domain of that programming system. The apparent simplification is clear; the file system and translation arrows have disappeared, as has the (semantically superfluous) copy of the graph of objects. In a nutshell, this has been achieved by extending the boundary of the programming system semantics beyond that of shared and long-term data.

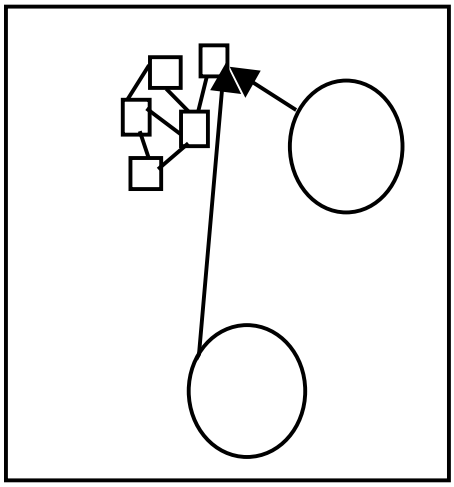

Fig. 2. The persistent application environment

The most obvious advantage of persistence is that the programmer is no longer burdened with the requirement of writing code to flatten and unflatten data structures which have to be passed into a different semantic environment in order for their long-term storage to be effected. The flattening code is time-consuming to write and sensitive to maintain, as it typically relies upon the semantics of an external system which is subject to change.

Although this was one of the initial motivations of the work, this is now viewed as relatively insignificant compared to the later advantages found through more advanced research. Such code is relatively straightforward to write, and many modern programming systems provide some support for the process; for instance the “pickle” operator in Modula-3 [CDG+89] and the “persistence” support in Metrowerks Codewarrior [MW98]. Persistence starts to show its true power in contexts where it is difficult or impossible to write such code, such as high-level languages where values significant to the semantics of a program are not necessarily denotable. Prime examples of these constructs are first-class closures and abstract data types, and the

reader is referred to the literature to find examples of the power of persistence in such contexts [POS8, POS7, Con90].

3.2. Persistence and the Web

The observation here is that, as applications which require to process data from around the Web become more sophisticated, they will start to encounter the same class of problems as those identified at the start of persistence research. Communicating processes around the Web can address each other directly, rather than using the intermediary of a file system[1]; however all current Web protocols are based on the transfer of files around the network. This gives almost the same scenario as sharing through file systems, except that the logical namespace ranges over a different set of values. The scenario is as pictured in Figure 3; the Web processes may be able to name each other, but are still required to perform the translation code and, perhaps more importantly, to keep semantically incongruous copies of the data associated with each invocation.

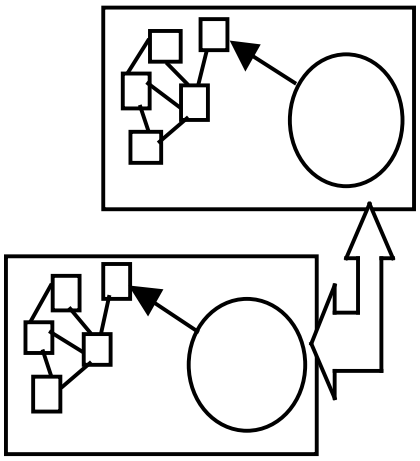

Fig. 3. Sharing data among Web processes

The topic of investigation is therefore whether the same solution, that of extending the boundary of data handled in the semantic domain of a programming language to include that of all process invocations, is viable in the context of the Web. At the start of persistence research it was clear that this approach was semantically preferable, but not that such systems were feasible to engineer and use. In the context of the Web, even the desired semantics of such an arrangement are less straightforward.

---

[1] This is ignoring the fact that the naming service of the Web is essentially derived from file system conventions. In this description we assume that URLs name resources orthogonally to their type, and may be used for example to name data channels between processes.

The good news is that, if a useful semantic model exists, then much of the research committed to persistent systems should be applicable to the new domain. Many of the engineering problems that have been successfully solved in closed persistent systems have clear parallels in the Web environment, and the implementation problems following a clear model might not be so great. The rest of this paper categorises the classes of problem which occur in the semantic integration, and outlines some areas in which possible solutions may be found.

4. A model for persistence and the Web

In its most general manifestation, orthogonal persistence is a property of a programming language whereby the treatment of data is entirely orthogonal to its lifetime. One model of this is to replace traditional file system access primitives by two language meta-constructs, *intern* and *extern*. Given a value *x*, of type *t*, and a global external namespace ranged over by *n*, then

```
extern( n, x )
```

causes a conceptual link to the value *x* to be placed in the global namespace, such that the call

```
intern( n, t )
```

results in the same value, formerly denoted by *x*. *intern* will fail dynamically if the identifier *n* does not denote a binding of type *t* in the external namespace. The crucial property of this model is that the semantics of the value is unaffected, even when concepts such as identity and mutability are in the semantic model.

The observation that the *intern* and *extern* calls can take place in different program invocations gives rise to the sharing of typed data among programs in different contexts. This single, shared namespace which allows any program invocation to access the same data roots is the key to orthogonal persistence.

It should also be noted that the notation of *intern* and *extern*, although used consistently since the inception of persistence research, is not necessarily helpful as it gives the intuition of a namespace external to the programming system domain. In reality these meta-functions give a mechanism whereby the data that hangs from the namespace is internal to the semantics of the programming system as a whole: thus the namespace is not in any sense external, but in fact shared internally by all program invocations.

We have chosen the intern/extern model as a starting point because it is the simplest model which, by its addition to an arbitrary programming language, provides orthogonal persistence. In this sense it is a core model, rather than a useful model for programming persistent applications, where more support is generally considered desirable. However we assert that if this model can be extended to an understandable semantics in the domain of Web data, then other more programmer-friendly models can also follow.

Our aim is then to apply this model of persistence in the domain of the internet, and more specifically the Web. Three main conflicts have been identified:

- the Web and the persistence model each have their own namespaces, which must therefore be unified
- the Web and any programming language each have their own preconceptions about the representation and structure of data, which must again be unified
- the Web has an (implicit) data model whose semantics necessarily allow the description of remote, unreliable, and autonomous data; desirable programming systems include notions such as static typing and referential integrity, which appear to be fundamentally at odds with this.

Each of this is now examined in turn.

4.1. Unification of the namespaces

The simplest way to achieve unification with the global persistent namespace assumed by the intern/extern model involves compromising it by the use of two different namespaces, that of a local file system and that of URLs. *extern* is used with the local file system namespace, and there are two forms of *intern*, one using the local file system namespace and one using the URL namespace.

In this way, the namespaces used by the proposed system are in fact identical to those used by the writers and readers of standard HTML documents in the domain of a web server. These are written into locations (filenames) resolved in the context of the local file system, and read as either local filenames or URLs, depending on the context and purpose of the reader. The author of an HTML document needs to understand both namespaces to allow the placement of anything other than simple relative resource names within the HTML code.

The use of these namespaces to share data between program invocations requires knowledge of the implicit mapping between them. For example, the local filename

```
/users/staff/richard/public_html/fred.html
```

and the URL

```
http://www.dcs.gla.ac.uk/~richard/fred.html
```

might map to the same physical file, but this is not normally specified by any formal arrangement. This mapping must for the moment remain beyond the semantics of the persistent system, and the programmer's knowledge of it must be assumed. Once again, the situation is identical with the human user's view of the namespaces.

Having simply established the namespaces, their use for the storage and retrieval of persistent data looks simple. To give a concrete example, the following code creates a new object of class *person* and stores it in an external namespace within the domain of an *http* server:

```
fred = new person( "Frederick", 31 )

extern(   "/users/staff/richard/public_html/fred.html",
          fred )
```

This object may subsequently be retrieved by a remote application executing the following code, and the identifier *thisFred* is typed locally as *person*.

```
thisFred =
internURL(
          "http://www.dcs.gla.ac.uk/richard/fred.html",
          person )
```

If the data is not accessible, or has some other type, the *internURL* operation fails and the binding is not made.

Readers unfamiliar with orthogonal persistence may at this point become suspicious about the simplicity of this code, and make guesses about the semantics of these statements. It is important to note that the definition of orthogonal persistence requires that the values associated with the bindings *fred* and *thisFred* in the example are semantically indistinguishable; preserving this is a serious implementation problem, but one that is now known to be tractable. Note also that another definitional feature of orthogonal persistence is that values of any type in the semantic domain are allowed the full range of persistence. Although the type *person* might be relatively straightforward to handle, the operations *intern* and *extern* are available on values of any type, including for example functions, closures and abstract data types.

However our task currently is to provide a clean semantic model for persistence and the Web. Although such a model is clearly of no use if it is not possible to implement, we do not discuss issues of implementation further in this context.

One unfortunate effect of the two-namespace model is to lose the nice property of context-free name resolution within the persistent system, as the resolution of local filenames depends upon the physical location in which the application is executing. One of the major perceived advantages of closed-world persistent systems is that the semantics of an application is entirely independent of its physical context, depending only on how the persistent store is populated.

This model relies upon a single global namespace, which is believed by many to be impossible to engineer, and some compromise is necessary in a system which extends beyond a local context. It is perhaps ironic that the URL model provides precisely this, but in reality this is only viable because of the update restrictions that exist with it. An alternative approach, for example using the URL namespace as a parameter to *extern*, could provide context-free name resolution, but the significant behaviour of the application would nonetheless necessarily depend upon protection issues based on the context of execution. It is pleasing at least that the intern namespace is a single global model, a feature which has so far evaded the engineers of closed persistent systems.

One final point that must be stressed is that it is only the concept of namespace that has been discussed in this section. With many Web and file system protocols, URL naming directly corresponds to the presence of physical files which contain the data being referenced; this is not necessarily the case in the system we are describing. The names describe only entry points into a general graph of persistent data, and not necessarily the physical location of a file in which the data is stored. Ultimately, of course, resource names are interpreted according to the whim of the http server that happens to be listening at the specified socket; however, to keep within the initial brief, it would be helpful to assume a standard server arrangement.

4.2. Representing persistent values

To give the most useful unification of persistence with the Web it is a requirement that documents created to represent persistent values are not only transmitted via the standard protocols, but can also make sense outside the context of the persistent system. It is a major feature of the internet, and undoubtedly one of the reasons for its success, that documents are in open standard formats and may be interpreted by a choice of viewers. We therefore avoid the choice of inventing a new document standard which would require all users of persistent data to adopt our technology.

That choice leaves the further question of how best to use existing and emerging standards to represent persistent values. There are currently a plethora of new standards emerging which allow better specification of Web resources than simple HTML. For example the W3C [W3C] standard XML, in conjunction with a DTD and XLL, could be used to describe a way of laying out text files to capture the semantics of values in any particular type system. Such an approach would seem unexciting, however; although in principle such resources would be re-usable by other systems, they would effectively be closed outside the persistent environment as a very high degree of interpretation would be required to allow their semantics to emerge. Claiming that a system using such protocols would be open would be similar to claiming any other system is open if viewed at the level of its base implementation.

The Resource Description Framework [RDF] allows the association of meta-data, loosely comparable to programming language types, with Web resources. Even if a high-level type system could be modeled, however, the granularity of the abstraction would appear to be too coarse-grained for use in a persistent Web system. The fundamental cost of remote resource fetching is generally believed to be in the connection cost, rather than bandwidth limitation, and it would appear that the fine-grained value model of persistent programming would not map well to every persistent value being represented as a separate resource. The model also suffers from the same problem of interpretation identified with the XML approach.

Another emerging standard of particular interest is that of semi-structured data. This data format emerged, amongst other uses, as a data interchange format, and some recent research has concentrated on the derivation of structure and semantics from this common level description [NAM98, NUW+97]. It may turn out to be possible to store persistent data in a semi-structured arrangement automatically derived from the static language typing, in such a way that its type-safe re-introduction can be automated [CS98c]. The beauty of this approach is that the semi-structured data is effectively self-describing, and its interpretation does not depend upon understanding a meta-level description. Therefore it could be used by any other application system capable of understanding semi-structured data. If this approach is tractable, it may also be possible to safely import semi-structured data with a different derivation into a typed persistent computation. However there are some fundamental research issues to be solved before this approach can be deemed tractable.

For the moment, it was decided to take a purely pragmatic approach simply to test the concept in an open and highly available manner. Entry points to persistent data are represented as HTML files, in a way that can be usefully interpreted by Web browsers as well as by a persistent program. This is achieved by creating HTML documents that contain persistent data formatted in a human-readable manner,

according to their type, via a simple set of rules. This allows programs to communicate with each other directly using typed data, and also allows this data to be understood by a human using a standard browser. It also provides the simplest of mechanisms for an applications builder to place the results of programs on the Web.

The necessary non-human-readable content, such as the canonical type representations required for run-time structural checking, are placed immediately before the data in HTML comment fields. This information can then be used when a persistent binding is specified using an *intern* statement, and the HTML code which represents the data can be properly interpreted according to its type. This arrangement also allows other strategies to be used in cases where the efficiency of the parsing might be an issue, for example a comment field might contain a compressed format of the same data, or information on how to fetch it from a different physical location.

The rules required to translate from typed values to human-readable versions are relatively straightforward to describe for most types. The reverse mapping is less so, but given that this can be guided by the embedded type representation it presents no real technical problems. Types such as closures and abstract types present more of a challenge; however previous research on hyper-programming [KCC+92] can be used to solve many of the problems very neatly, by providing a model of closure that bears a textual representation. The issues of protection in abstract data types should also be noted, although it seems unlikely that there is a perfect solution. This is maybe one of the fundamental type system compromises that will have to be made.

Figure 4 gives an example of the HTML generated by the execution of the following program fragment:

```
fred = new person( “fred”, 31 )
extern( “/users/staff/richard/public_html/fred.html”,
          fred )
```

```
<HTML>
<HEAD>
...
</HEAD>
<BODY>
<!--HIPPO TYPE ANNOTATION
t0=str(age*name#int*string)
-->
<TABLE BORDER="2">
<TR><TD COLSPAN=2>Structure of type Person</TD></TR>
<TR><TD>name</TD><TD>age</TD></TR>
<TR><TD>Fred</TD><TD>23</TD></TR>
</TABLE></BODY>
</HTML>
```

Fig. 4. Example HTML generation

An important point to notice is that such files need be generated only at the execution of a call to *extern*. The semantics of these calls is to generate only entry points into arbitrary graphs of data, rather than having the effect of storing a flattened closure as in a non-persistent system; what is outlined above, therefore, is not a data storage paradigm, but only a way of representing entry points. Data storage may be handled in a relatively traditional manner by a persistent object storage mechanism, provided that the illusion is somehow preserved of the same data being accessed either from a local persistent system, from a remote persistent system, or from a Web browser. Furthermore, accesses in the last two categories should generate the same text streams through the http protocol.

The problems left to handle are those of object identity (including the correct semantics of mutable objects), and the representation of references to other objects.

Object identity can not be equated with URL equality, the latter including aliases that are in general undetectable. Instead, however, unique identities can be manufactured by the persistent run-time system, as is common in any case, and stored along with type information as an invisible field. On intern, the identity information can be handled by the persistent store layer of a run-time system to avoid duplication. The fact that the identity is not observable to HTML browsers is actually desirable, as it fits with that apparent semantic model. Preserving correct update semantics poses a further problem, which will be dealt with in the next section as the proposed solution is based upon a typing mechanism. Given this can be handled, however, this approach gives a level of flexibility in that semantically identical values can safely have multiple representations, in more than one text file.

There is an easy conceptual solution to the problem of representing references in the dual format. It is necessary for them to be represented as URLs in a textual view

of the data, to allow human browsing of the graph of objects accessible from the persistent entry point. Given this, all that is required is to generate a new text file to represent each reachable persistent object, generate a unique local filename for it, and plug a corresponding URL into the textual description of the referring object. The disadvantage of this approach, as mentioned before, is that transmitting each persistent object with a separate http request seems to be an inappropriate level of granularity. A solution to this exists however in the use of HTML anchors, which allow arbitrary numbers of objects to be transmitted in a single stream. Rather than pre-calculate these multi-object files, it is possible to embed unresolved references as CGI calls based on the identity of the referred object, allowing them to be dynamically generated. Given that we can solve the problem of the same semantic values appearing in multiple text files, this mechanism can be used strategically in conjunction with established techniques such as object clustering to send an appropriate set of values through each connection.

4.3. A type system for Web-based persistence

Even given that standard HTML documents can be used to represent typed data, there are still two problems with persistent applications which operate over such data. These are due to fundamental mismatches between the semantics of programming languages and the normal model of Web usage:

- referential integrity, normally guaranteed within a programming system, can not be expected in general in this domain
- many programming models include update, with locations contained within arbitrary data structures; however, most data provided on the web is read-only, and not allowed to be updated in an arbitrary manner by users other than its provider

We conclude with brief descriptions of type system mechanisms we have introduced to combat these problems.

4.3.1. Typing for referential integrity

A great deal of research in persistent programming systems has concentrated on the subject of dynamic binding, often referred to as flexible, incremental binding. The binding of code to persistent data, captured above through the *intern* operation, requires a check both on the existence and the type of the data referred to. The consensus model to emerge is that the existence of the data does not require to be individually checked, and can be folded into the same language mechanism as the type check. This is safe, as a failed type check prevents any further execution on the value referred to, although not general, as the programmer can not distinguish between the two classes of failure. However experience with persistent programming has not shown any convincing requirement to do this. In the new domain of the internet our initial assumption is that any dynamic unavailability of data is best exposed to the programmer of a persistent application as a dynamic type error. It should be noted that the fundamental differences in the context may turn out to invalidate this assumption.

The problem is easily solved by resorting to a type system where the each use of an object is dynamically checked. This was the approach taken in the first persistent

programming language, PS-algol [ABC+83]. However further work on persistent languages showed that a much better solution could be achieved, with the addition of explicit dynamic typing constructs to an otherwise statically checked system [MBC+89, Con90].

The semantics of *internURL* includes a dynamic check on the availability and type of the data, and this provides a failure point in the case where the data is unavailable or of the wrong type. The problem is that after such a check has succeeded, the data is integrated with the ongoing typed computation in a context where further dynamic checks are not normally desirable. The only way to ensure the soundness of the type system is to make a locally cached copy of any data integrated in this manner.[2]

It is impossible in general to cache the transitive closure of data fetched from a particular entry point in the persistent graph, as this could itself form a global web of data. We have therefore designed a type system which distinguishes between data which is known to be available and that which may be dynamically unavailable. It is important to note that this does not by itself solve the problems of closure and referential integrity, but is proposed merely as a platform on which to conduct experiments with the relationship. Any final solution must include some element of compromise between purity of semantics and a pragmatic approach to the downloading of data files, so at the moment we strive only to have at least some definition of data movement within the programming language domain.

The type constructor *refR* (remote reference) is added to an otherwise standard type system to denote the type of data which is not guaranteed to be available. *refR*(*t*) is, in general, a supertype of *t*, and a dynamic coercion is required to use the denotation as type *t*. The semantics of this type are similar to that of that of the type *any* in Napier88 [MBC+89] and *dynamic* in [Car85, ACP+95], except that an indication of the value's expected type is included with its static denotation. The typing of a denotation as *refr*(*t*) gives no guarantee whatsoever about the type of the value referred to, and in this sense contains no static type information whatever. However a value of such a type has the specialisation operation to *t* defined over it, and before type-specific computation can take place the type must undergo an explicit dynamic check. If the check succeeds then the value is cached at that time and can be subsequently relied upon during the computation.

Although the typing *ref*(*t*) does not carry a guarantee, it is however the case that where the link to the data is dynamically available, and where the data has not been interfered with by any other autonomous system, then the dynamic specialisation will succeed. In this sense the *t* can be regarded as a static hint to the real type of the data, and the programmer may have some useful model about the probability of success.

The reference type mechanism allows the programmer fine-grain yet implicit control over the point at which data is fetched. For example

```
thisFred = internURL(
          "http://www.dcs.gla.ac.uk/richard/fred.html",
          refR( person ) )
```

[2] For the purposes of this description we ignore the issues of cache coherency, although this does present a major problem. Our current thinking is to handle this at a different level from the programming model which we describe here. The prevention of remote update, as described later, greatly alleviates the general problem.

causes an association to be made between the identifier *thisFred* and the URL, but does not cause the data to be fetched and cached, whereas this statement

```
thisFred =internURL(
          "http://www.dcs.gla.ac.uk/richard/fred.html",
          person )
```

causes the data to be fetched also. In a less trivial example,

```
internURL(
          n,
          structure( x : int; y : refR( person ) ) )
```

allows the programmer to specify the fetch of the top-level structure, but not necessarily its closure. The subtyping relationship between *person* and *refR*(*person*) allows the type check to succeed only when the record originally placed at that binding contained a person or any subtype of person. If the closure is fetched for pragmatic reasons, such as the object clustering referred to earlier, any later dynamic checks can be elided.

Although this construct gives the potential for applications which perform arbitrary dynamic typing, and which may therefore fail at any point with a type error, it is nonetheless possible to adopt a coding style where all dynamic checks are factored out at the start of a application. This can give all the desirable properties of static typing whilst retaining the required flexibility of type-safe remote binding.

It is our final hope, at this point completely untested, that such a typing strategy might be integrated with remote object batching strategy, as hinted at above, in a way which is able to allow robust distributed applications with clear semantics.

4.3.2. Typing for remote update

Update causes a final problem in the context of a persistent language operating over data available from the internet. There is one easy solution, which is to claim that traditional query languages are essentially declarative, and any such language used to query the internet should be a declarative subset of the general persistent system. While this solves the problem of programs specifying updates to remote data that they do not own, we prefer if possible to handle the problem without losing the ability to update data that is owned locally.

There are two problems with this model in the context as described so far. The first is that multiple file representations of objects that contain mutable locations may potentially lead to inconsistent views of a set of data. The second is that unprotected remote update is not generally acceptable in the context of the internet, and some protection is required to prevent this from being specified within computations.

A location type is used to control the problem of generalised network update. The Hippo language has an explicit type constructor, *loc*(*t*), which must be used to denote a mutable location. Furthermore, the system contains a subtyping rule that states for any type *t*, *loc*(*t*) is a subtype of that type. This is a fairly standard subtyping arrangement where mutability is explicitly typed (see for example [Car89]).

A corollory of such typing is that, for any type *t* which contains location types, there exists another type *s* which does not contain location types, and of which *t* is a subtype. A denotation typed as *s* has the operational ability to dereference any

locations contained in any value typed at *t*; however, the same value typed as *s* does not allow any update operation to its locations. We refer to *s* as the immutable supertype of *t*.

When remote data is typed at an *intern* command, the dynamic test for success is that the statically asserted type is an immutable supertype of the type associated with the physical data. This gives that static knowledge that it is impossible to specify a well-typed computation that causes an update to occur to a value which is not locally resident.

This mechanism does not solve the generalised problem of cache coherency over the programming system, although it may make it considerably more tractable. In particular a fundamental attribute of the subtyped location model is that values which are not typed as locations may nonetheless change by side-effect, and this is the required semantics in this case. Two possibilities are under investigation, details of which are beyond the scope of this paper; one is to build a re-fetch model into the *refR* type scheme, and the other is to impose a causal consistency model over the networked persistent processes. The latter of these, whilst obviously more complex, may turn out to fit neatly with other aspects of persistent Web computation that are not discussed here [CS98b].

5. Conclusions and future work

This paper gives only an introduction to the key concepts used in the Hippo language in terms of its intention to integrate an orthogonally persistent programming system with the World-Wide Web. The overall task is greater than those problems and outline solutions presented here, and further information about the project is available from the Web site[3] and other recent publications [CS98a, CS98b]. It should be stressed that this is a position paper from an early stage of the project, and whilst the ideas contained in this paper are believed to be robust, they are not immune to change.

The methodology underlying the project is strongly directed towards implementing robust programming systems that can be used to program Web-based applications, to therefore achieve maximum feedback as to useful design. To this purpose programming systems are available to researchers who might be interested in writing applications in the domain; please contact the authors for details.

6. Acknowledgements

Richard Connor is supported by an EPSRC Advanced Research Fellowship B/94/AF/1921, and Keith Sibson by an EPSRC studentship. Some of the work described is based on earlier work supported by EPSRC grant GR/K79222. Paolo Manghi holds a Marie Curie TMR award.

[3] http://www.hippo.org.uk/

We would also particularly and sincerely like to thank an anonymous referee who made many helpful comments on unclear areas and unspoken assumptions in the original text.

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.

[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.

[ACP+91] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. “Dynamic typing in a statically-typed language” ACM Transactions on Programming Languages and Systems, 13(2):237-268, April 1991

[ACP+95] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy “Dynamic typing in polymorphic languages” Journal of Functional Programming, 5(1):111-130, January 1995

[AJDS97] M.P. Atkinson, M. Jordan, L. Daynès and S. Spence “Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System” *in Persistent Object Systems - Principles and Practice*, Connor and Nettles (Eds), Morgan Kaufmann, 1997 pp 33-47.

[AM95] M.P Atkinson and R Morrison *Orthogonally Persistent Object Systems* VLDB Journal 4, 3 pp 319 - 401

[Atk78] Atkinson, M.P. “Programming Languages and Databases”. In Proc. 4th IEEE International Conference on Very Large Databases (1978) pp 408-419.

[Car85] Cardelli, L. “Amber”. AT&T Bell Labs, Murray Hill Technical Report AT7T (1985).

[Car89] L. Cardelli "Typeful Programming" DEC SRC Technical Report No. 45 (May 1989)

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Digital Equipment Corporation Systems Research Center, 1989.

[Con90] Richard Connor “Types and Polymorphism in Persistent Programming Systems” PhD Thesis, University of St Andrews, 1990.

[CS98a] Richard Connor and Keith Sibson “Paradigms for Global Computation - an Overview of the Hippo Project” Proc. Workshop on Internet Programming Languages, in conjunction with the IEEE Computer Society International Conference on Computer Languages 1998 (to appear).

[CS98b] Richard Connor and Keith Sibson “HCL - the Hippo Core Langauge”Proc. Workshop on Internet Programming Languages, in conjunction with the IEEE Computer Society International Conference on Computer Languages 1998 (to appear).

[CS98c] Richard Connor and Fabio Simeoni “SSSub – a subtyping system for abstracting over semi-structured data.” Submitted for publication.

[GN98] Garthwaite A. and Nettles S. “Transactions for Java” in Proc. 1998 International Conference of Programming Languages, May 1998 pp 16-27.

[Hip98] The Hippo Project Homepage, http://www.hippo.org.uk/

[KCC+92] Kirby G.N.C., Connor R.C.H., Cutts Q.I., Dearle A., Farkas A. & Morrison R. "Persistent Hyper-Programs" Proc. 5th International Workshop on Persistent Object

Systems, San Miniato, Italy, September 1992, in **Persistent Object Systems, San Miniato 1992**, Springer-Verlag, pp 86 - 106.

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. "The Napier88 Reference Manual". University of St Andrews Technical Report PPRR-77-89 (1989).

[MW98] Metrowerks Codewarrior, http://www.metrowerks.com/

[NAM98] Nestorov S., Abiteboul S. and Motwani R Extracting Schema from Semistructured Data Proc SIGMOD'98, SIGMOD Record 27, 2, June 1998 pp 295 - 306

[NUW+97] Nestorov S., Ullman J., Wiener J and Chawathe S. Representative Objects: Concise Representations of Semistructured, Hierarchical Data Proc ICDE, Birmingham, UK 1997, pp 79 - 90

[POS7] *Persistent Object Systems, Tarascon 1994* Atkinson M., Maier D. and Benzaken V. (Eds), Springer-Verlag Workshops in Computer Science, 1995.

[POS8] *Persistent Object Systems - Principles and Practice*, Connor R. and Nettles S. (Eds), Morgan Kaufmann, 1997

[PS88] "PS-algol Reference Manual, 4th edition". Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[RDF] http://www.w3c.org/RDF/

[SCW85] Schaffert, C., Cooper, T. & Wilpot, C. "Trellis Object-Based Environment Language Reference Manual". DEC Technical Report 372 (1985).

[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.

[W3C] http://www.w3c.org/