HYPER-PROGRAMMING WEB APPLICATIONS

Joshua D. Fox [1], Henry Detmold [1] & Katrina Falkner [1]

[1] School of Computer Science,
The University of Adelaide

ABSTRACT: The Web has become the dominant platform for the development of a large class of multi-user applications. In contrast to conventional approaches, current Web application development technology places little emphasis on system enforcement of static program safety properties. This departure from historical developments has the result that users of Web applications are forced to endure failure modes that would never be accepted from conventional applications. This paper describes work addressing this problem by applying a technology known as hyper-programming to the development of Web applications. This approach provides a number a program safety benefits. For example, not only does our system prevent 404 errors for hyperlinks between parts of an application, but it also guarantees that the action attached to forms take parameters that are consistent with the form inputs, in terms of name, number and type. In this paper we describe HyperWeb, a prototype implementation of the hyper-programming concept applied to Web application development. This system provides Web applications with system enforced program safety properties. Enforcement of program safety properties begins at program composition time (coding) and is continuously maintained thereafter, thereby promoting improved Web application quality and providing an enhanced user experience.

INTRODUCTION

When companies rely on Web applications, to interact with customers and communicate with partners, even small coding mistakes can have serious repercussions. To achieve successful results, Web application developers must adopt the best practices of conventional software development processes, e.g. nightly build systems combined with error prevention techniques such as strict coding standards, functional testing, regression testing, and monitoring. However, in the Web development context, these processes operate largely in the absence of system enforced static program safety guarantees, which are a cornerstone of conventional development approaches. The result of this lack of system enforced program safety is that Web application failure rates tend to be significantly higher than for conventional applications of equivalent complexity [oSF03a, oSF03b].

This project exploits a novel application development paradigm, hyper-programming, which addresses this lack of program safety. Furthermore, this style of programming provides a set of static safety guarantees, which can be established even while the program is being composed. A hyper-programming style system, HyperWeb, has been built to evaluate the benefits and applicability of this style of system for the construction of Web applications. A summary of work related to safe Web application development is presented. We describe the hyper-programming concept. Models of both Web application failure modes and of safe Web applications are developed, in order to evaluate HyperWeb. We evaluate HyperWeb by describing its use, specifically in relation to the failure modes and safety model previously identified. We explore the design and implementation of HyperWeb, by extension of the WebStore Web application server, and conclude with a summary of our contribution.

RELATED WORK

Most current approaches to safety of Web application development are client-based. Few existing approaches have adopted a server-side model, and hence the advantages obtained from server side safety are not achieved. Relevant systems with similar goals to that of HyperWeb include <bigwig> [BMSS99] and the W3Objects [ICL96] system.

<bigwig> is a high-level programming language for developing interactive Web services. <bigwig> aims to remove some of the obstacles that face current developers of Web services in order to lower cost of production and maintenance, and increase functionality and reliability over CGI/Perl scripts, ASP,

and PHP based solutions. Services are session based: Web services are not viewed as collections of pages or scripts, but as potentially complex sequences of interactions between servers and clients.

In <bigwig>, HTML is a built-in data type. HTML templates are first-class values that may be computed and stored in variables. An HTML template may contain named *gaps* that are place-holders for other HTML templates or for text strings. Gaps may at runtime be plugged with concrete values, which may also contain gaps, providing a dynamic mechanism for building Web documents. Compile time type checking ensures that documents are used in a consistent manner: for instance, that the form inputs in the document match the server code that receives the values. Compile time analysis ensures that only valid HTML 4.01 documents are constructed and sent to clients at runtime.

<bigwig> addresses several failure modes such as malformed content, inconsistencies in links to Web services and failed session state linkage. However, <bigwig> does not enforce link integrity. Furthermore, <bigwig> suffers from increased complexity as a result of considering each failure mode in isolation.

The W3Objects system provides low-level referencing mechanisms to address the broken-link failure. Web resources are represented as objects (W3Objects), which are encapsulated resources possessing internal state and well defined behaviour. Inter-object references, such as those representing hyperlinks, are maintained by objects in the form of *smart references*. Through the use of a forward referencing scheme, smart references remain valid even in the event of the migration of the target object. The scheme also provides an information provider with useful knowledge of which objects are currently externally referenced. This knowledge ensures that referenced objects are not inadvertently deleted, and acyclic garbage is identified. Although the W3Objects approach addresses link integrity, it does not address other failure modes such as malformed pages or inconsistencies in links to Web services.

HYPER-PROGRAMMING

Conventional application development paradigms provide developers with a set of static safety guarantees through compilation and linking stages. However, this set is incomplete and weak due to the following limitations not addressed by this methodology:

1. Access specification checks for dynamically loaded objects are not performed until runtime. The static safety guarantees achievable for objects that can be fully resolved at compile time can thus not be attained for dynamically loaded objects. Typically, transient data and most program code is statically checkable, whereas persistent data (which pre-exist and outlive a program's execution) and library code is dynamically loaded and so access to such objects is not statically checkable. Therefore, the scope within which the static checking regime operates is incomplete.
2. There is no guarantee that a textual description of an object's access specification will remain valid until the time of its resolution — even if it is valid at the time of composition, compilation, or linking. Programs are typically constructed and stored in some long-term storage facility, such as a file system, separate from the run-time environment (which terminates its existence at the end of each program execution). Such storage facilities allow their topologies to be manipulated by external users, including the deletion or moving of code and data. These actions result in an object's access specification becoming invalid. Three possible scenarios are:

   - The object is moved so that the access path no longer resolves to it, even though it still exists.
   - The object is deleted.
   - The object is replaced with one of a different type.

   Hence, static safety guarantees provided by conventional approaches are weak - they are conditional guarantees dependent on users adhering to policies avoiding these scenarios.
3. Even where checking is static, it still applies after the program has been written - during compilation and linking stages. Since it is known that the cost of fixing problems is diminished when those problems are found earlier, a better option is to provide checking while the program is being composed. Early work recognising this led to syntax directed editors and other such tools.

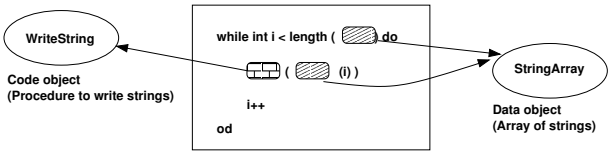

Figure 1: Example Hyper-program

Static guarantees on program safety can be achieved through means other than compilation and linking. In fact, these guarantees can be made stronger, be achieved earlier than compilation, and can also be achieved for dynamically loaded persistent objects providing a more complete set of static safety guarantees. These advances have been realised in systems implementing *hyper-programming* [KCCM93].

Hyper-programming is a style of programming suitable for persistent systems - those involving the use of data that outlives program execution. In an orthogonally persistent system [AM95, MCK+99] programs are constructed and stored in a persistent store, the same environment in which they are executed. This means that the (persistent) objects (both code and data) accessed by a program may already be available at program composition time. In this case, actual pointers to the objects can be directly included in the program source instead of textual access paths as depicted in Figure 1. One of the properties of a persistent system is that pointers are guaranteed to be valid for the duration of their lifetime.

A program containing both text and actual pointers to objects is known as a hyper-program [KCCM93]. The use of hyper-programming with a persistent language provides a number of benefits including: increased ease of program composition, strong static safety guarantees, and the ability to perform access checking for dynamically loaded persistent objects as early as program composition. The last benefit is of particular interest as it appears to provide a solution to the limitations of current Web application development paradigms. Early access checking eliminates the runtime occurrence of access failures including the absence of a required object, and situations where an access path resolves to an object of an unexpected (and unacceptable) type. In the Web application development context, the components of a Web application (pages etc.) are effectively dynamically loaded persistent objects accessed via textual access paths (URLs). Therefore, Web applications are highly apposite examples of the kinds of applications for which hyper-programming was originally intended.

A MODEL OF WEB APPLICATION FAILURE MODES

Recent studies [oSF03a, oSF03b] by The Business Internet Group of San Francisco (BIG) attempt to quantify the integrity of Web applications on 40 industry leading commercial Web sites and 41 Web sites under the management of the U.S. Government. The findings indicate Web application failure rates of 72.5% and 68% respectively. Such failure rates are not acceptable in conventional applications, and yet are being tolerated in Web applications. The BIG study categorises Web application failures as follows:

- Technical Errors: Technical errors involve application server and Web server errors. For instance, blank pages, 404 file not found, 500 internal server error and embedded content errors.
- Incorrect Data Errors: Incorrect data errors involve programming, database, and human errors. For instance, wrong page returned or wrong item returned.
- User Failure Errors: Errors that prevent the user from completing a transaction without the presence of an error message. For instance, being unable to purchase after product comparison because the wrong shipping address was selected.

Incorrect Data Errors and User Failure Errors are typically the result of human error for which there is no automated solution. Our concern is with Technical Errors, many of which can be avoided in programming languages through the provision of referential integrity (safe pointers) and type safety. As such, we view technical errors in Web applications as program failure modes. We introduce a model of Web application failure modes from a human user's perspective [DFM+03]. Our model identifies four failure modes:

1. Malformed page failures: these occur when pages containing invalid HTML are served by the Web server. In such cases, the pages do not operate as intended with the user's Web browser; possibly they cannot be rendered and thus the user is denied full access to the intended functionality.

2. Broken link failures: these occur when the destination of a link within an application is erroneously deleted or moved, or when the application generates a page containing a link that does not refer to an extant object. These failures lead to users experiencing the well known '404 - Not Found' failure.

3. Inconsistencies in links to Web services: the ACTION attribute of an HTML form associates the inputs in the form with a Web service responsible for servicing those inputs. The form and service can vary independently, but are related by an implicit requirement for consistency which must be maintained. In particular, the form must ensure the submission of all inputs expected by the service and their type correctness. If a service is updated and becomes inconsistent with the form(s) referencing it (or vice versa) then users will experience failures when they submit the form(s).

4. Failed session state linkage: since session state is not statically scoped, a session-oriented application can fail during execution due to the absence of required state. Many systems mask this failure, by creating the missing session state with default values and with the type implied by the point of use. This type may be inconsistent with the use in other parts of the application. This masking effect leads to erroneous application behaviour, which, while less severe in appearance, is in fact more difficult to diagnose and correct than the underlying failure.

A MODEL OF WEB APPLICATION PROGRAM SAFETY

In order to avoid the failure modes listed previously, our system guarantees that the corresponding causal errors do not occur. It does so through the enforcement of the following properties:

1. Ensuring all delivered HTML content is syntactically well-formed.

2. Ensuring referential integrity of hyper-links in both static and dynamically generated content.

3. Ensuring consistency of Web forms with the services processing form input.

4. Ensuring statically safe binding of session operation code to variables defined with session scope.

Of course, the devil is in the detail as to how these properties are achieved! Essentially, our approach proceeds by analogy to properties in conventional programming approaches. We first identify safety properties in conventional programming that are analogous to the properties listed above. Then, we transplant a solution enforcing the conventional program safety property to the Web context.

Ensuring all delivered HTML content is syntactically well formed is analogous to ensuring that any value returned by a function call is of the correct type. In HyperWeb all Web programs return object graphs, which are values of a type that is isomorphic to well formed HTML. A system level process transforms object graphs into text. Correctness of this process ensures that any user level Web program can only generate syntactically valid HTML, enforcing the first safety property. Of course, broken browsers may still not display valid HTML correctly, however, this is beyond the scope of this project.

Ensuring referential integrity of hyper-links (including dynamically generated hyper-links) is analogous to the maintenance of referential integrity in programming languages that support safe pointers. In such languages, reclamation of the storage allocated to an object is performed by a system-level garbage collection process. This ensures that objects are only ever reclaimed when they are un-referenced. Modern programming languages such as Java use garbage collection in combination with strict rules for assignment and initialisation of pointers to ensure that any pointer that could in the future be de-referenced will resolve to the intended object. In HyperWeb we represent the entire Web application as an object graph, with sub-graphs representing pages and (internal) hyper-links represented by pointers. System level processes provide a bi-directional mapping between this representation and Web content (HTML

etc.). Since referential integrity is maintained on the object representation, and the Web representation is isomorphic to the object representation, referential integrity is maintained on the Web content, enforcing the second safety property.

To ensure consistency of Web forms with the associated services (ACTION pointers), we first note that action pointers are like any other hyper-link and referential integrity is ensured as described above. Second, we observe that calling a service to process form inputs is analogous to a procedure call in programming, and specifically that the requirement for inputs to match those expected by the service is analogous to the requirement that actual parameters match formal parameters. Accordingly, we model Web forms as a set of *Input* objects and a pointer to an *Action* object, which represents the service. Action objects are essentially first-class functions that take form Input objects as parameters, and as such, type correctness of form inputs is assured, enforcing the third safety property.

Finally, we view ensuring statically safe binding of the code of session operations to variables defined with session scope as analogous to ensuring the binding of higher order functions to variables within a containing lexical scope. Specifically, if a function *inner* is defined within a function *outer*, then *inner* has access to the local variables of *outer*. Then, if the return value of an activation of *outer* includes a reference to *inner* within its closure (as is permitted if functions are higher-order), it will be possible to call the returned *inner* function outside of an activation of *outer*. To support such calls, languages supporting higher-orders implement block retention retaining the activation record of *outer* after it has returned. Notice that scope is completely static, the variables of *outer* that *inner* accesses are known to be both defined and of fixed type, for any call to *inner*. Application of this idea to session scope variables is relatively straightforward given the preceding parts of the model. The operation defining the entry point to a session is a function defining the session state as local variables and subsequent operations within the session as lexically nested functions. These functions have static access to the session state, enforcing the fourth safety property.

A further potential source of errors is the execution of JavaScript. However, this problem is left as further research while we address the previous issues.

USING THE HYPERWEB SYSTEM

HyperWeb is a Web application for the construction of Web applications. Web applications are constructed via a high level interface, or hyper-code representation, as shown in Figure 2. For pragmatic reasons, this hyper-code representation renders Web pages in a fashion similar to the rendering by the end user's Web browser. Each element is augmented with additional components that provide the hyper-code editing functionality. The hyper-code representation of a Web page consists of a two column table with rows representing the HTML elements nested within the <BODY> of that page. The right cell in each column renders the corresponding BodyElement object. The left cell contains information about the corresponding BodyElement object, and options allowing objects to be inserted, manipulated, and have the pointers to them removed. All static linking and type checking is incorporated into the composition process requiring no separate compilation or linking stages.

When inserting, a list of possible types that are valid for inclusion at the selected location are presented to select from. This ensures Web pages precisely model the containment rules of valid HTML. For example, a cell of a table is not permitted for insertion into a the Body of an HTML document. When replacing an object, the type is chosen implicitly by the system to be the type of the object being replaced. This ensures objects are only ever replaced with objects of the same type. It is important to note that these are system level assertions — enforced by HyperWeb's type system and not (just) by a tool. Web applications are constructed with objects that are created and stored on a persistent server providing type safety and referential integrity properties on these objects at time of composition, and continuously maintaining them for the duration of the application's lifetime. In contrast, a (client-side) tool can not possibly ensure that objects wont be deleted, moved, or replaced with objects of a different type once the application is deployed or in subsequent (or concurrent) application evolution.

When either inserting or replacing objects, new objects can be created, or extant objects be reused. Creating a new object involves entering values to instantiate the object. For example, when creating a new Table the number of rows and columns and the table header can be entered. Alternatively, the

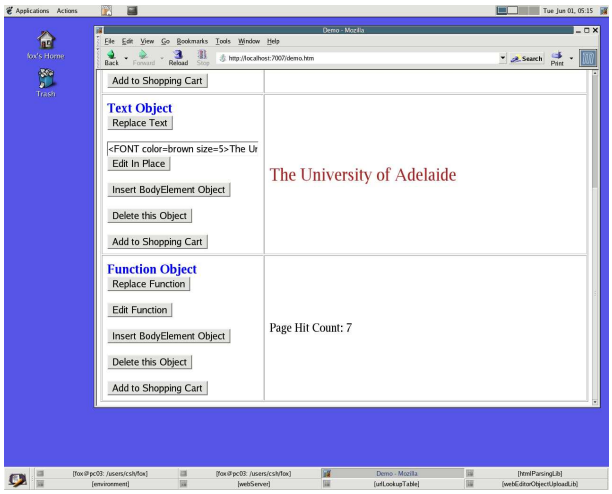

Figure 2: Screen shot of Hyper-code Representation

author may browse existing Web page object graphs or shopping carts within the system. In this case, extant objects of the chosen type may be reused and imported by reference or by copy.

It is often useful to embed dynamic content within otherwise static pages, a page hit counter is one example. As such, in each case that an object of a given type is permitted, a reference to a function object returning that type is also permitted. The type system ensures that dynamic content will always be syntactically valid. This facility allows the inclusion of dynamically generated content at fine granularity.

DESIGN AND IMPLEMENTATION OF HYPERWEB

The delivery of Web applications, as well as HyperWeb's interface, uses an extension of the Web-Store [DFM+03] Web application server. Figure 3 illustrates the life cycle of an HTTP request for Web content. Each Web page is represented as a directed acyclic graph of objects rooted at a document base object, HTMLObject (8). HTMLObjects are wrapped in typed first class functions, or *content generators* (7), that may be parameterised to accept parameters sent in an HTTP request (1). Content generators are stored under URIs in a URI lookup table (4). URIs are represented as typed first class functions, or *request handlers* (5): Request handlers are generated for content generators as they are inserted into the lookup table under a specified URI. The request handler's function is to accept an HTTP request with the required parameters expected by the content generator. The request handler extracts these parameters from the request, and calls the content generator with the parameters as arguments (6). The request handler is then added to the lookup table under the specified URI.

When the Web server receives an HTTP request string (1) for this URI, the request string is parsed (2) into an HTTPRequest object (3). The HTTPRequest object is then used as an argument in a call to the request handler (5) for that URI. The request handler in turn calls the content generator (7) with the necessary parameters (6) extracted from the HTTPRequest, and the requested HTMLObject is produced (9). The serving of content from within the requested HTMLObject's object graph requires conversion from its structured form to its textual form. This is referred to as the *un-parse and serve* process (10). The un-parse and serve process is analogous to object serialisation. A depth first traversal of the object graph is performed, and for each object that corresponds to an HTML element the appropriate HTML starting tag is output pre-traversal of its children, and the matching end tag is output post-traversal. During this process, information stored within each object relating to its HTML representation is output appropriately. As a result, a string of HTML (11) is produced in response and served to the client via standard HTTP. Given that the un-parsing process produces correct HTML text for any HTMLObject object graph, the application will generate only valid HTML output. As such, malformed page failures are statically prevented and we achieve the first safety property: ensuring all content is well-formed.

HyperWeb extends the type system of WebStore to provide a richer programming environment including

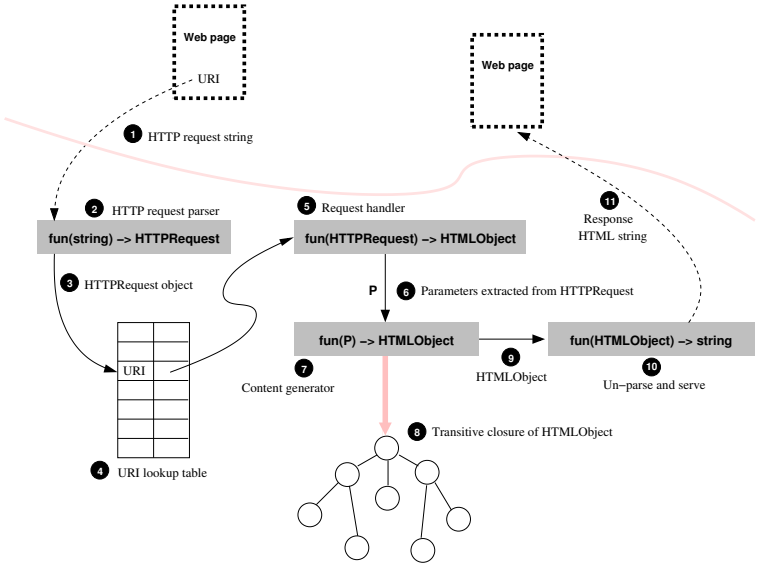

Figure 3: WebStore's request life cycle

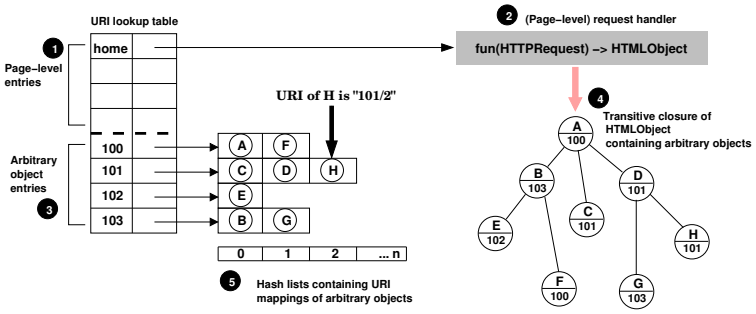

Figure 4: Mapping Objects of Arbitrary Type to Unique URIs

elements such as Web forms. All content is represented in a structured form. Each Web page is represented as a directed acyclic graph of objects rooted at a document base object, HTMLObject. The HTMLObject type models the structure of well-formed HTML so it is a Cartesian product containing a head field and a body field. This modelling applies transitively and so the BodyObject type is a container for objects corresponding to valid body-level content, such as <A HREF> and <FORM> elements.

HyperWeb's principal function is the manipulation of object graphs. To support this, each object involved in a manipulation must be accessible by the functions performing the manipulation. These functions are provided as Web services. In order for the Web service to access an object, each object is assigned a unique URI and is located via a URI lookup table similar to that used by WebStore. WebStore's URI lookup table only stores RequestHandler objects for content generators that return (page-level) HTMLObjects. HyperWeb makes extensions to WebStore's lookup table to include objects of arbitrary type. In this way, a single lookup table is used for every object in the system, be it a first class function, HTML element object, or data structure.

URIs for page-level content generators can be chosen by the author so they can be easily identified. However, all other objects have URIs generated for them. Storing string representations of each object's URI is not space efficient. Thus, when an object is to be added to the URI lookup table, it is given a pseudo random integer, referred to as its *hash URI*. As such, the lookup table is extended to include entries for *hash lists* in which objects with the same hash URI are stored. HyperWeb's URI lookup table is illustrated in Figure 4.

One example of the use of these URIs is by the forms in the editing cell of an object's hyper-code representation. Suppose we wish to replace an object with another of the same type. The URI of the object's *location* is embedded in a hidden input in the 'Replace Object' form. When the author selects 'Replace Object', this URI is posted to a service allowing the user to browse for the replacement object. Once located, the editing cell of the replacement object also contains a form with that object's URI embedded in a hidden input. Selecting this form sends both URIs, the location's and the replacement's, to a service which then performs the replacement. Even if the topology of the object graph is altered within this time, the URIs remain valid.

Given guarantees on access to objects via URIs, HyperWeb supports a range of safe and consistent operations for the creation, manipulation and replacement of any kind of object within the system. An important subset of these operations involve Web forms and their corresponding Web services. These operations include the ability to edit the parameters of a form's (higher-order) Action object, and have the form's set of Input objects automatically updated to remain consistent, and vice versa.

CONCLUSION

The Web is rapidly becoming the dominant platform for the construction of large multi-user applications. However, in the rush to the Web, many of the traditional techniques for improving the quality and reliability of applications have been abandoned. Specifically, static guarantees of program safety such as type correctness and referential integrity are not ensured. As a result, users of Web applications are forced to endure failure modes that would never be accepted in conventional applications. HyperWeb provides the first proof of existence of the application of the hyper-programming concept to the construction of Web applications. The interface to HyperWeb is generated entirely within the confines of standard HTTP and is therefore accessible via any Web browser. HyperWeb is not simply another tool-based approach to Web application development. HyperWeb is a server-based approach in which Web applications are constructed out of objects that are created and maintained on a persistent server. As such, static program safety guarantees are provided and maintained for the duration of the application's lifetime.

REFERENCES

[AM95] M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3):319–401, 1995.

[BMSS99] C. Brabrand, A. Møller, A. Sandholm, and M. I. Schwartzbach. A runtime system for interactive Web services. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1391–1401, 1999.

[DFM+03] H. Detmold, K. Falkner, D. Munro, T. Olds, R. Morrison, and S. Norcross. An integrated approach to static safety of web applications. In *The 12th International World Wide Web Conference (WWW03)*, 2003.

[ICL96] D. Ingham, S. Caughey, and M. Little. Fixing the "Broken-Link" problem: the W3Objects approach. *Computer Networks and ISDN Systems*, 28(7–11):1255–1268, 1996.

[KCCM93] G. N. C. Kirby, Q. I. Cutts, R. C. H. Connor, and R. Morrison. The implementation of a hyper-programming system. Technical Report CS/93/5, PhD thesis, University of St Andrews, 1993.

[MCK+99] R. Morrison, R. C. H. Connor, G. N. C. Kirby, D. S. Munro, M. P. Atkinson, Q. I. Cutts, A. L. Brown, and A. Dearle. The napier88 persistent programming language and environment. In M. P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*, pages 98–154. Springer, 1999. ISBN 3-540-65772-X.

[oSF03a] The Business Internet Group of San Francisco. The black friday report on web application integrity, January 2003. www.tealeaf.com/downloads/news/analys_report/BIG-SF_BlackFridayReport.pdf, viewed 30th March, 2004.

[oSF03b] The Business Internet Group of San Francisco. Government web application integrity, January 2003. www.tealeaf.com/downloads/news/analyst_report/BIG-SF_Report_Gov_2003-05.pdf, viewed 30th March, 2004.