Domain Specific Languages for Interactive Web Services

Claus Brabrand

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Domain Specific Languages for Interactive Web Services

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Claus Brabrand
November 7, 2002

Abstract

This dissertation shows how domain specific languages may be applied to the domain of interactive Web services to obtain flexible, safe, and efficient solutions.

We show how each of four key aspects of interactive Web services involving sessions, dynamic creation of HTML/XML documents, form field input validation, and concurrency control, may benefit from the design of a dedicated language.

Also, we show how a notion of metamorphic syntax macros facilitates integration of these individual domain specific languages into a complete language.

The result is a domain specific language, `<bigwig>`, that supports virtually all aspects of the development of interactive Web services and provides flexible, safe, and efficient solutions.

Acknowledgments

First of all, I would like to thank Michael Schwartzbach for supervising my Ph.D. and especially for bringing my attention to the Ph.D. program.

A special thanks go to my office mate, colleague, and good friend, Anders Møller.

I thank my undergraduate study group: Tom Sørensen, Thomas Hune, and Flemming Friche Rodler, and the entire `<bigwig>` team; in particular Anders Sandholm, Mikkel Ricky, and Steffan Olesen.

I also thank the BRICS research center for providing an inspiring and truly international environment; in particular the following *set* of BRICS people: `{` Jesus Almansa, Marco Carbone, Olivier Danvy, Uffe Engberg, Jesper Gulmann, Martin Lange, Paulo Oliva, Rasmus Pagh, Pawel Sobocinski, Frank Valencia, Mads Vanggaard, Maria Grazia Vigliotti `}`.

I would also like to thank IBM Research for the three valuable months I spent there. Thanks go to my manager, Roger Pollak, my mentor, John Ponzo, my colleagues Kristoffer Rose and Phillipe Audebaud, and my office mates.

I thank the people I got to know from studying one year in Strasbourg: in particular, Jacob Grydholt and Patricia d'Erneville.

Furthermore, I thank all my friends.

Last, but not least, many thanks to my mom, Lise Krause, my dad, Keld Brabrand, and brother, Mads Brabrand, and to the rest of my family.

Tak / thanks / merci / grazie / danke / jërë jëf,

Claus Brabrand,
Aarhus, November 7, 2002.

Contents

Part I

Context

List of Publications

- **The `<bigwig>` Project**
  with Anders Møller and Michael I. Schwartzbach.
  Transactions on Internet Technology (TOIT), Vol. 2, No. 2, pp. 79–114, ACM, May 2002.

- **A Runtime System for Interactive Web Services**
  with Anders Møller, Anders Sandholm, and Michael I. Schwartzbach.
  In Proceedings of the Eighth International World Wide Web Conference (WWW8), pp. 313–324, Elsevier, May 1999.
  Also in Journal of Computer Networks, Vol. 31, No. 11–16, pp. 1391–1401, Elsevier, May 1999.

- **PowerForms: Declarative Client-Side Form Field Validation**
  with Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach.
  In World Wide Web Journal, Vol. 3, No. 4, pp. 205–214, Baltzer Science Publishers, December 2000.

- **Static Validation of Dynamically Generated HTML**
  with Anders Møller and Michael I. Schwartzbach.
  In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Programming Analysis for Software Tools and Engineering (PASTE'01), pp. 38–45, ACM, June 2001.

- **Language-Based Caching of Dynamically Generated HTML**
  with Anders Møller, Steffan Olesen, and Michael I. Schwartzbach.
  In World Wide Web Journal, Vol. 5, No. 4, pp. 305–323, Kluwer Academic Publishers, 2002.

- **Growing Languages with Metamorphic Syntax Macros**
  with Michael I. Schwartzbach.
  In Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02), pp. 31–40, ACM, January 2002.

Chapter 1

Introduction

1.1 Domain Specific Languages for Interactive Web Services

This dissertation shows how *domain specific languages* can be applied to the domain of *interactive Web services* to achieve *flexible*, *safe*, and *efficient* solutions.

We will examine this thesis by breaking down the domain of interactive Web services into relatively independent aspects. Each of these aspects will then be analysed and addressed by its own dedicated domain specific language. We will also show how each of these resulting sub-languages are integrated into a language, `<bigwig>`, for developing interactive Web services. Finally, we will show how this integration can be supported by a notion of *metamorphic syntax macros*.

First, however, we will briefly introduce the two concepts of the title: *domain specific languages* and *interactive Web services*.

1.2 Domain Specific Languages

Domain specific languages (DSL) are unlike general purpose languages (GPL) designed to write a particular kind of programs.

Of course, domain specific languages do not add expressive power beyond Turing completeness; any program that can be written in a DSL can also be written in a GPL. Even so, domain specific languages have many advantages over general purpose languages.

The paramount advantage is that the level of abstraction can be made to correspond directly to that of the problem domain. Concepts inherent to the problem domain can be turned into abstractions in the DSL.

Although similar abstractions can be defined in libraries of general purpose languages, they must still be used in the full context of the GPL with all the details of parameter mechanisms, scope rules, and so on. Furthermore, they are limited to the abstraction mechanisms and invocation syntax of the GPL. In contrast, domain specific languages may be given any syntax that more directly reflects the idiom of the problem domain. Also, certain problematic constructions may be explicitly prohibited in the syntax.

Domain specific languages also permit sophisticated domain specific analyses which may be used to restrict usage or as a basis for optimization. Comparatively, this can never be achieved with a library in a GPL. In general, nothing prevents the programmer from misusing a library, for instance by calling certain functions in the wrong order.

Domain specific languages are often declarative which make them easier to read, write, and modify. They are often simple enough to be used by non-programmer domain experts. To this end, programs are more concise, almost to the point of being self-documenting in that they embody directly the knowledge from the domain.

Domain specific languages really only have one, yet considerable, disadvantage: the cost of construction. Their realization often requires many iterations of analysis, design, implementation, and evaluation. However, once created, they increase productivity, reliability, and maintainability.

We refer to [89] for more information on domain specific languages, including advantages over general purpose languages.

1.3 Interactive Web Services

The HyperText Transfer Protocol, HTTP, was originally designed for browsing *static* HTML documents on the World Wide Web. The need for up-to-date and customized documents spawned the creation of the Common Gateway Interface, CGI, which is a platform-independent method for creating documents *dynamically* based on client input. To this end, HTML was equipped with a collection of standard input widgets for selecting and entering various kinds of data. The notion of *interactive Web services* is obtained by appropriately sequencing such client interactions. In our work, we focus on interactive Web services, which are Web servers on which clients can initiate sessions that involve several exchanges of information mediated by HTML forms.

We have identified the following relatively independent key aspects of interactive Web services that must be addressed in all realistic services:

- *sessions:* clients must be guided appropriately through interactions while retaining state;
- *dynamic documents:* HTML documents must be constructed dynamically;
- *form field validation:* data entered by clients must be validated;
- *concurrency control:* session processes run in parallel which means that concurrency aspects must be dealt with;
- *database integration:* most services employ a database that must be integrated; and
- *security:* Web services are inherently distributed which means that various security aspects must be addressed.

We have analysed the first four of these aspects and for each of them designed a domain specific language targeted uniquely for that particular domain.

1.4 Structure of the Dissertation

Chapter 2 introduces the concept of a *session*, presents the main challenges along with our solution; a runtime system, `Runwig`. The runtime system is futher explained in the paper [16] which can be found in Chapter 11.

Chapter 3 presents the `DynDoc` language [72] for dynamically constructing HTML/XML documents which forms a basis for the next two chapters. Chapter 4 shows how the `DynDoc` language can be analysed to statically guarantee that only *valid* HTML documents are ever shown to a client. This result is from the paper [17] which is included in Chapter 13. Chapter 5 then shows how the static parts of dynamically generated documents can be cached on the clients. This solution was the topic of the paper [14] which can be found in Chapter 14.

Chapter 6 presents our sub-language for addressing form field input validation, `PowerForms`, which was presented in the paper [15] and which is included in Chapter 12.

Chapter 7 describes the concurrency control sub-language [71, 13], `SyCoLogic`.

Chapter 8 introduces a notion of *metamorphic syntax macros* which may be used to integrate the many sub-languages in the `<bigwig>` language. This macro language is further explained in the paper [19] which is found in Chapter 15.

Chapter 9 concludes by demonstrating how the domain specific languages have obtained flexible, safe, and efficient solutions for each of their domains. It is also shown how these sub-languages are integrated into the `<bigwig>` language[1] for developing interactive Web services. The `<bigwig>` language is presented in the paper [18] which is included in Chapter 10.

[1]See the `<bigwig>` project homepage: `http://www.brics.dk/bigwig/` for documentation and implementation.

Chapter 2

Sessions

2.1 Introduction

Most interactive Web services today are implemented using a *single interaction paradigm* wherein the focus is on a single interaction with a client. Conceptually, there is one program per interaction and a whole service is constructed as the appropriate sequential composition of essentially independent programs. Such a program is executed upon request from a client, receiving form data as input and producing HTML as output before terminating. Individual requests are tied together by inserting appropriate links to other programs in the reply pages. Thus, a Web service is defined by a collection of loosely related programs. This process is illustrated in Figure 2.1.

A major problem with this approach is that the flow of control is implicit. The overall behavior of a service is distributed over numerous individual programs and depends on the implicit manner in which they pass control to each other. This design complicates maintenance in that it is hard to identify which programs together form a service and how they are related. The design also precludes any sort of automated global analysis, leaving a whole class of errors to be detected in the running service [34, 3]. In particular, there is no way

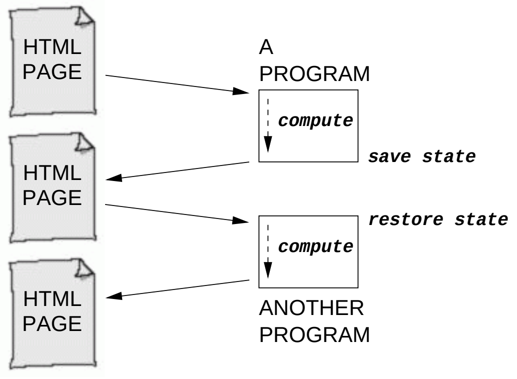


Figure 2.1: An interactive Web service specified as the sequential composition of two essentially independent programs. On the left is the client's browser, on the right are the two programs running on the server.

of checking interaction correspondence of programs; that is, whether the form input fields in the output of one program correspond to those expected as input to the next.

Another major problem is handling local state. While persistent data shared among all session threads is stored naturally in a database, data local to a particular session or sequence of interactions has to be managed explicitly. Since individual programs terminate between interactions, the local state must somehow be passed on to the next programs. One solution is to pass the local session data via the client to subsequent programs in hidden input fields, in cookies, or encoded as part of the URL. However, these three approaches all store the state on clients which has some obvious security implications in that it may be tampered with or contain sensitive information that should not be disclosed. Another solution is for a program to explicitly save the state on the server before it terminates so that it can be reloaded and restored by the next program. In any case, the programmer needs to deal with these low-level issues of handling and serializing the state.

The single interaction paradigm can be divided into two main approaches: the *script-centered* and the *page-centered*. Each is supported by various tools and suggests a particular set of concepts inherent to Web services. The two approaches will be briefly outlined in the following.

The Script-Centered Approach

The script-centered approach builds directly on top of the plain, stateless HTTP/-CGI protocol. An interaction is specified in the form of a program written in some general purpose programming language obeying the CGI protocol for receiving input and producing output. Input in the form of textual data from form input fields is decoded from a special environment variable, `QUERY_STRING`, or from standard input, depending on the submission method used for invoking the script. HTML output is typically created on the fly using `print`-like statements.

A prototypical scripting language is Perl, but almost any programming language has been suggested for this role. CGI scripting is often supported by a large collection of library functions for decoding form data, validating input, accessing databases, and realizing semaphores. Even though such libraries are targeted at the domain of Web services, the language itself is not.

The Java Servlets language also fits into this category. The overall structure of a service written with servlets is the same as for Perl. Every possible interaction is essentially defined by a separate script, and one must use cookies, hidden input fields, or similar techniques to connect sequences of interactions with the clients. Servlets provide a session tracking API that hides many of these details. Many servlet servers use cookies if the browser supports them, but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled. This API is exemplified by the following code inspired by two Servlet tutorials[1]:

[1]`http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/` and `http://java.sun.com/docs/books/tutorial/servlets/`

```
public class SessionServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    ServletContext context =  getServletContext();
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><HEAD><TITLE>Servlet Demo</TITLE></HEAD><BODY>");
    if (session.isNew()) {
      out.println("<FORM ACTION=SessionServlet>" +
                  "Enter your name: <INPUT NAME=handle>" +
                  "<P><INPUT TYPE=SUBMIT></FORM>");
      session.putValue("state", "1");
    } else {
      String state = (String) session.getValue("state");
      if (state.equals("1")) {
        String name = (String) request.getParameter("handle");
        int users =
          ((Integer) context.getAttribute("users")).intValue() + 1;
        context.setAttribute("users", new Integer(users));
        session.putValue("name", name);
        out.println("<FORM ACTION=SessionServlet>" +
                    "Hello " + name + ", you are user number " + users +
                    "<P><INPUT TYPE=SUBMIT></FORM>");
        session.putValue("state", "2");
      } else /* state.equals("2") */ {
        String name = (String) session.getValue("name");
        out.println("Goodbye " + name);
        session.invalidate();
      }
    }
    out.println("</BODY></HTML>");
  }
}
```

Clients running this service are guided through a series of interactions: first, the service prompts for the client's name, then the name and the total number of invocations is shown, and finally a "goodbye" page is shown. The `ServletContext` object contains information shared to all sessions, while the `HttpSession` object is local to each session. The code is essentially a `switch` statement that branches according to the current interaction. An alternative approach is to make a servlet for each kind of interaction. In spite of the API, one still needs to explicitly maintain both the state and the identity of the session.

The model of sessions that is supported by Servlets and other script-centered approaches tends to fit better with "shopping basket applications" where the client browses freely among dynamically generated pages, than with complex services that need to impose more strict control of the interactions.

The Page-Centered Approach

The page-centered approach is covered by language such as ASP, PHP, and JSP, where the dynamic code is embedded in the HTML pages. In a sense, this is the inverse of the script-centered languages where HTML fragments are embedded in the program code. When a client requests a page, a specialized Web server interprets the embedded code, which typically produces additional HTML snippets while accessing a shared database. In the case of JSP, implementations work by compiling each JSP page into a servlet using a simple transformation.

This approach is often beautifully motivated by simple examples, where pages are mainly static and only sporadically contain computed contents. For example, a page that displays the time of day or the number of accesses clearly fits this mold. The following JSP page dynamically inserts the current time together with a title and a user name based on the CGI input parameters:

```
<HTML><HEAD><TITLE>JSP Demo</TITLE></HEAD><BODY>
Hello <%
  String name = request.getParameter("who");
  if (name==null) name = "stranger";
  out.print(name);
%>!
<P>
This page was last updated: <%= new Date() %>
</BODY></HTML>
```

The special `<%...%>` tags contain Java code that is evaluated at the time of the request. As long as the code parts only generate strings without markup it is easy to statically guarantee that all shown pages are valid HTML and other relevant properties. But as the services become more complex, the page-centered approach tends to converge towards the script-centered one. Instead of a mainly static HTML page with some code inserted, the typical picture is a single large code tag that dynamically computes the entire contents. Thus, the two approaches are closely related, and the page-centered technologies are only superior to the degree in which their scripting languages are generally better designed.

The ASP and PHP languages are very reminiscent of JSP. ASP is closely tied to Microsoft's Internet Information Server, although other implementations exist. Instead of being based on Java it defines a language-independent connection between HTML pages and scripting languages, typically either Visual Basic Script or Microsoft's version of JavaScript. PHP is a popular Open Source variant whose scripting language is a mixture of C, Java, and Perl.

These languages generally provide only low-level support for tracking client sessions and maintaining session state. Cookies, hidden input fields, and some library support is the common solution. Also for other Web service aspects, such as databases and security, there is often a wide range of libraries available but no direct language support.

2.2 The Session-Centered Approach

The pure session-centered approach was pioneered by the MAWL project. A service is here viewed as a collection of distinct *sessions* that access some shared data. A client may initiate a session *thread*, which is conceptually a process running on the server. Interaction with the client is viewed as remote procedure

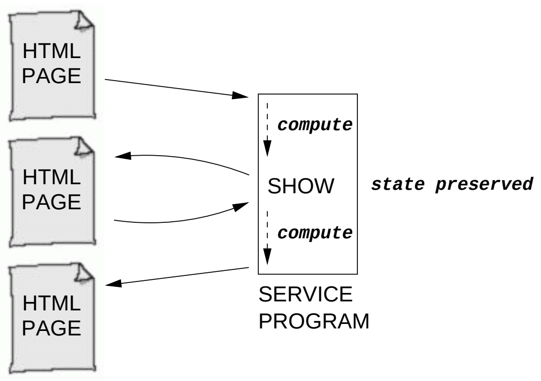


Figure 2.2: Client-server sessions in Web services. On the left is the client's browser, on the right is a session thread of the service program running on the server. The thread is initiated by a client request and controls the sequence of interactions.

calls from the server, as known from classical construction of distributed systems but with the roles reversed.

The flow of an entire session is programmed as a single sequential program, which is closer to ordinary programming practice and offers the compiler a chance to obtain a global view of the service. Figure 2.2 illustrates the flow of control in this approach. Important issues such as concurrency control become simpler to understand in this context and standard programming solutions are more likely to be applicable.

The following MAWL program is equivalent to the previous Servlet example:

```
static int users = 0;

session GreetingSession {
  auto form {} -> {handle} hello;
  auto string name = hello.put().handle;

  auto form {string who, int count} -> {} greeting;
  users++;
  greeting.put({name, users});

  auto form {string who} -> {} goodbye;
  goodbye.put({name});
}
```

The HTML templates *hello*, *greeting*, and *goodbye* are placed in separate files. Here is `hello.mhtml`:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Enter your name: <INPUT NAME=handle>
</BODY></HTML>
```

and `greeting.mhtml`:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Hello <MVAR NAME=who>, you are user number <MVAR NAME=count>
</BODY></HTML>
```

The template for `goodbye` is similar. A form tag and a continue button are implicitly inserted. Variables declared `static` contain persistent data, while those declared `auto` contain per-session data. The `form` variables are declared with two record types. The former defines the set of gaps that occur in the template, and the latter defines the input fields. In the templates, gaps are written with `MVAR` tags. Template variables all have a `put` method. When this is executed, the arguments are inserted in the gaps, the resulting page is sent to the client who fills in the fields and submits the reply, which is turned into a record value in the program. Note how the notion of sessions is explicit in the program, that private and shared state is simply a matter of variable declaration modifiers, and that the templates are cleanly separated from the service logic. Obviously, the session flow is more clear, both to the programmer and to the compiler, than with the non-session based approaches. One concrete benefit is that it is easy to statically check both validity and correct use of input fields.

The main force of the session-centered approach is for services where the control flow is complex. Many simple Web services are in actuality more loosely structured. If all sessions are tiny and simply does the work of a server module from the page-centered approach, then the overhead associated with sessions may seem to large. Script-centered services can be seen as a subset of the session-centered where every session contains only one client interaction. Clearly, the restriction in the script-centered and the page-centered languages allow significant performance improvements. For instance, J2EE Servlet/JSP servers employ pools of short-lived threads that store only little local state. For more involved services, however, the session-centered approach makes programming easier since session management comes for free.

2.3 Structure of <bigwig> Services

The overall structure of `<bigwig>` programs is directly inspired by MAWL. A `<bigwig>` program contains a complete specification of a Web *service*. A service contains a collection of named *sessions*, each of which is essentially an ordinary sequential program. A client has the initiative to invoke a thread of a given session, which is a process on the server that executes the corresponding sequential code and exclusively communicates with the originating client. Communication is performed by *showing* the client an HTML page, which is implicitly made into a form with an appropriate URL return address. While the client views the given document, the session thread is suspended on the server. Eventually the client submits the form, which causes the session thread to be resumed and any form data entered by the client to be *received* into program variables. A simple `<bigwig>` service that communicates with a client as in the Servlet and MAWL examples is the following:

```
service {
  html hello = <html>Enter your name: <input name="handle"></html>;

  html greeting =
    <html>Hello <[who]>, you are user number <[count]></html>;
  html goodbye = <html>Goodbye <[who]></html>;

  shared int users = 0;

  session Hello() {
    string name;
    show hello receive[name=handle];
    users++;
    show greeting<[who=name,count=users];
    show goodbye<[who=name];
  }
}
```

The program structure is obviously as in MAWL, except that the session code and the templates are wrapped into a `service` block. The `show-receive` statements produce the client interactions similarly to the `put` methods in MAWL. However, `<bigwig>` provides a number of new features. Most importantly, HTML templates are now *first-class values*. That is, `html` is a built-in data type, and its values can be passed around and stored in variables as for any other data type. Also, the HTML templates are *higher-order*. This means that instead of only allowing text strings to be inserted into the template gaps, we also allow insertion of other templates. This is done with the special *plug* operator, x`<[`y`=`z`]` which inserts a string or template z into the y gaps of the x template. Clearly, this constitutes a more flexible document construction mechanism, but it also calls for new ideas for statically verifying for instance HTML validity. This is the topic of Chapter 3 and 4. Other new features include the techniques for improving form field validation and concurrency control, together with the syntax macro mechanism, all of which are described in the following chapters.

2.4 A Session-Based Runtime Model

The session-based model can be implemented on top of the CGI protocol. One naive approach is to create session threads as CGI scripts where all local state is stored on disk. At every session interaction, the thread must be started again and restore its local state, including the call stack, in order to continue execution. A better approach is to implement each session thread as a process that runs for the whole duration of the session. For every interaction, a tiny transient CGI script called a *connector process* is executed, acting as a pipe between the Web server and the session process. This approach resembles FastCGI [66] and is described in detail in [16]. Our newest implementation is instead based on a specialized Apache server module[2]. Naturally, this is much faster than the CGI solutions since it does not create a new process for every single interaction, but only for the session processes. The runtime system is also available as a stand-alone implementation called `Runwig`[3].

Two common sources of problems with standard implementations of sessions are the history buffers and the bookmarking features found in most browsers.

[2]See `http://httpd.apache.org/`.

[3]available from `http://www.brics.dk/bigwig/runwig/`.

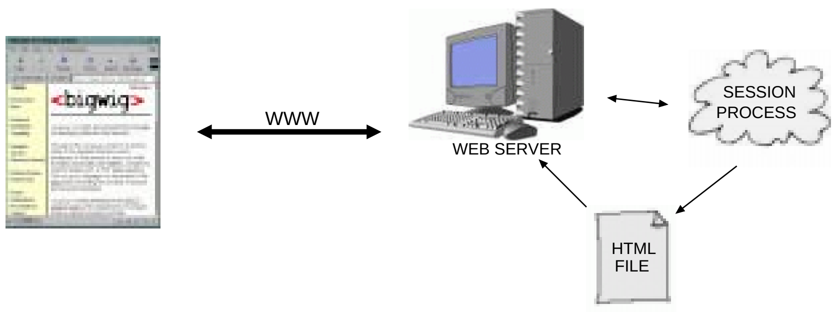


Figure 2.3: Session-based runtime model with reply indirection. Each session thread is implemented as a separate process that writes its HTML reply to a designated file.

With the history buffers and the *back button*, the users can step back to a page from a previous interaction, and either intentionally or unintentionally resubmit an old input form. Sometimes this can be a useful feature, but more often this causes confusion and annoyance to the users who may for instance order something twice. It is a general problem that the information shown to the user in this way can be obsolete since it was tailor-made only for the exact time of the initial request. Since the information was generated from a shared database that may have changed entirely, it does generally not make sense to "step back in time" using the history buffer. This is no different from ordinary programs. The problem is not only with external side-effects. Even observation may have consequences as also known from quantum theory's "*modification by observation*"; being able to backtrack would render the high-score feature of the number guessing game obsolete. Even if the programmer has been aware of this stepping back in time and has added serial number checks, the history buffer will be full of URLs to obsolete requests. If the service really needs a "back" feature, it can be programmed explicitly into the flow of the sessions. It also becomes hazardous to try to use bookmarks to temporarily suspend a session. Invoking the bookmark will then typically cause a CGI script to be executed a second time instead of just displaying its results again.

`<bigwig>` provides a simple but unique solution to these problems: Each session thread is associated a URL which points to a file on the server containing the latest HTML page shown to the client. Instead of sending the contents directly to the client at every `show` statement, we redirect the browser to this URL, as illustrated in Figure 2.3. Since the URL serves as the identification of the session thread, this solves the problems mentioned above: The history list of the browser now only contains a single entry for the duration of the session, the sessions can now be bookmarked for later use, and in addition the session identity URL can be passed around manually—to another browser for instance—without problems. When using URLs instead of cookies to represent the session identity it also becomes possible for a single user to simultaneously run multiple sessions in different windows but with the same browser.

With this simple solution we can furthermore automatically provide the client with feedback while the server is processing a request. This is done by after a few seconds writing a temporary response to the HTML file, which informs the client about the status of the request. This temporary file reloads itself frequently in the client's browser, allowing for updated status reports. When the final response is ready, it simply overwrites the temporary reply file, causing the reloading to stop and the response to be shown. This functionality is provided by the `flash` construct which takes an HTML document, instruments it will the necessary automatic reloading, and writes it to the associated reply file. By default, the runtime system redirects the client after eight seconds, providing a message stating "Reply not ready yet. Please wait...". This simple technique may prevent the client from becoming impatient and abandoning the session.

The `<bigwig>` runtime system additionally contains a garbage collector process that monitors the service and shuts down session processes which have been abandoned by the clients. By default, this occurs if the client has not responded within 24 hours. The sessions are allowed to execute some clean-up actions before terminating.

2.5 Other Related Work

Guide [57] is a rule-based language belonging to the single interaction paradigm. A *context* is carried across the scripts and handles state, distinguishing between local, session, and global variables. However, these contexts have to be managed explicitly by the programmer.

WASH/CGI [85] is an embedded DSL for server-side Web scripting based on the purely functional programming language Haskell and, in particular, on its monads. Like Hanus's Curry library [40], it provides a session abstraction using a *callback* mechanism in which *submit handlers* are bound to submit buttons, permitting evaluation to continue at different points according to which button was depressed. It allows session to be backtracked by idempotent reevaluation of the submit handler given the local state which is stored in a hidden input field on the client. In addition to efficiency issues, storing the state on the client has some security issues; it is vulnerable to tampering and may expose sensitive information. For example, a high-score is easily achieved on the `GuessNumber` WASH/CGI game either by inspecting the local state or by backing up the session once the number is guessed.

Chapter 3

Dynamic Generation of XML Documents

3.1 Introduction

An important aspect of Web service development is the construction of HTML reply documents customized for individual clients upon request.

Currently, nearly all Web services construct document replies through either the *script-* or *page-centered* programming approaches. Although the two approaches appear fundamentally different, they are related by an interesting duality. In the script-centered approach, default is programming, but the programmer can escape to printing HTML via a `print`-like command; whereas in the page-centered approach, default is printing HTML and escaping to programming is available through special embedded scripting tags, such as `<% ... %>` in the case of JSP. In fact, this is exactly what happens when a JSP page is *lexically* transformed into a Java Servlet; HTML entities are wrapped in invocations of `out.print` and embedded code is inlined in the resulting Servlet program. That this is a purely lexical process is illustrated by the following legal JSP fragment:

```
<% if (use_bold) { %>                               if (use_bold) {
  <b>bold</b>                                         out.print("<b>bold</b>");
<% } else { %>                 ==>                  } else {
  <i>italic</i>                                       out.print("<i>italic</i>");
<% } %>                                             }
```

This example also shows that the structure is an illusion; we cannot reason about even the presence of HTML constituents independent of the scripting elements. This means that also in the script-centered approach, all parts of the reply document and not just the dynamic content must be assumed to come as the output of a Turing complete computation. The output document is essentially constructed as the concatenation of several dynamically constructed strings. Thus, it is hard to reason statically about resulting documents. Specifically, this way of constructing documents

- **precludes static validation**, as it is impossible to statically analyse the program to determine whether all possible documents produce valid HTML; and

- **prevents checking of interaction correspondence**, since it is impossible to statically determine which form input fields are present in an output document, there is no way of checking whether this corresponds to what is expected as input to the next interaction.

In addition to the absence of static safety, there are also some important limitations on flexibility. This way of constructing documents

- **intermixes designer and programmer aspects**, as bits and pieces of HTML markup and program code are scattered about the service program, making it difficult for designers and programmers to identify their respective parts and operate independently [29]; and

- **forces linear document construction**, as documents must be constructed linearly from the first `<html>` element to the last `</html>` element, as opposed to being composed from components in a more logical manner.

These are four fundamental limitations that current developers must be willing to accept.

The MAWL Approach

The MAWL [55] language, has addressed these issues by introducing a notion of first-order HTML templates. A MAWL template is a complete HTML document with a fixed collection of named variables which can be substituted with simple dynamically computed string values when the document is to be presented to a client. Templates are placed in separate files, completely separating the service code and HTML code permitting programmers and designers to operate more independently. It is also possible to issue the two static safety guarantees, as the template readily contains all the HTML markup of the reply document.

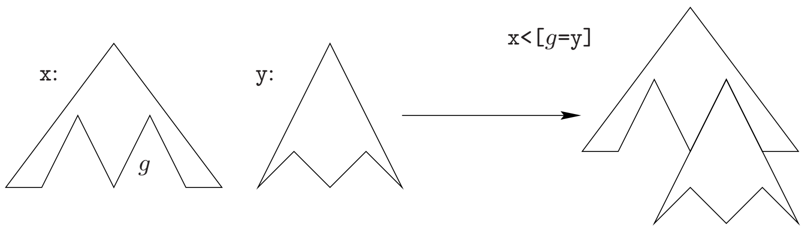
A disadvantage of this approach is that reply documents cannot be customized beyond a fixed number of simple parameterizations; only the dynamically inserted string data may vary. This is partially alleviated by a special iteration construction, `MITER`, that permits an unbounded list of simple values to be inserted into repetitions of an HTML fragment. However, this is insufficient to produce nested lists or tree structures.

3.2 A Language for Dynamic Generation of XML

In the following we will present a *flexible*, *safe*, and *efficient* language for dynamically generating XML documents that solves all these problems. We will now show how to regain the flexibility lost without compromising safety by generalizing the MAWL solution to *higher-order* templates.

Document Construction: The *plug* Operator

A document template constant is delimited by `<html>...</html>` and may in addition to normal HTML contain any number of named gaps; a gap named g is syntactically written as `<[`g`]>`. Documents are *first-class* values since they may be assigned, passed around, and stored in variables. A special *plug* operation is available for document construction. The expression, `x<[`g`=y]`, creates a new document value by inserting copies of `z` into all g gaps of a copy of `x`:



Since the documents designated by `x` and `y` may contain further gaps, this is a highly flexible and dynamic mechanism for constructing documents. It is reminiscent of higher-order functions, except that there is no alpha-conversion and only one implicit layer of binding gap names at the outermost level.

In full generality, the plug operation accepts full document expressions rather than document variables. The value plugged may also be a string or an integer in which case it is coerced to a document by converting any angled braces, "<" and ">" to "`&lt;`" and "`&gt;`", respectively; this ensures that markup only comes from the constant templates. Multiple gaps may be plugged in that `x<[`g_1`=y`$_1$`,`g_2`=y`$_2$`]` is syntactic sugar for `x<[`g_1`=y`$_1$`]<[`g_2`=y`$_2$`]`. We have also introduced a notion of *attribute gaps* which are gaps written inside elements that may provide an attribute with a dynamically computed value. Syntactically, they are written `<... a=[`g`] ...>`; where `a` is the name of the attribute and g the gap name. Of course, attribute gaps cannot be plugged with HTML, but only with string or integer values.

The following example gradually composes a "Welcome to BRICS" document.

```
service {
  html cover = <html>
    <head><title>Hi!</title></head>
    <body bgcolor=[color]><[contents]></body>
  </html>;

  html greeting = <html>Hello <[who]>, welcome to <[what]>.</html>;

  service {
    html h;
    h = cover<[contents=greeting];
    show h<[color="#9966ff", who="Stranger", what=<html><b>BRICS</b></html>];
  }
}
```

First, the ***`contents`*** gap of the template `cover` is plugged with the document `greeting` and assigned to `h`. This yields a new document with an attribute gap,

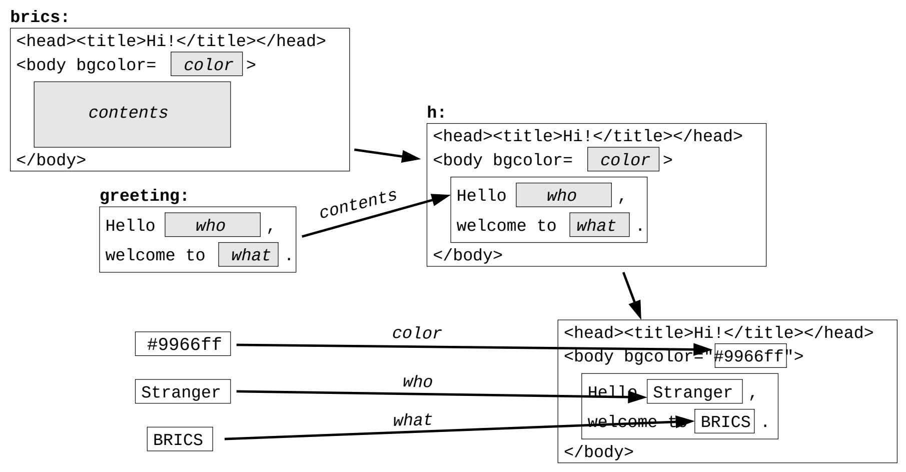


Figure 3.1: Building a document by plugging into template gaps. The construction starts with the five constants on the left and ends with the complete document on the right.

color from the original `cover` template and two HTML gaps, *who* and *what*, stemming from the `greeting` document. Then, these three gaps are plugged in succession and the result is shown to the client. The document construction process is illustrated in Figure 3.1

Client Interaction: The `show` Statement

Client interaction is provided by the `show` statement that takes a document value, implicitly plugs any remaining gaps with the empty string, shows it to the client, and suspends computation. The document is automatically wrapped with a `form` element with an appropriate `action` URL, so that when the client submits the document, it will reactivate the process with the same state as whence it paused. If not present, a default `submit` button is added, allowing the form to be submitted. The `show` statement may also have a `receive` part which provides a mechanism for receiving the values of form input fields into program variables. The following example illustrates the `show-receive` mechanism:

```
service {
  int n;
  string s;

  html input = <html>
    Name: <input type="text" name="name"><br>
    Age:  <input type="text" name="age">
  </html>;

  html output = <html>
    Hello <em><[user]></em>, <p>
    Next year, you will be <[next]> years old.
  </html>;

  session InputOutput() {
    show input receive[s = name, n = age];
    n++;
    show output<[user = s, next = n];
  }
}
```

It shows a document, `input`, prompting the client for name and age using two text input fields, *name* and *age*. When this document is submitted, the values entered are received into the program variable `s` and `n`. Hereafter, `n` is increased by one after which `s` and `n` are plugged into the `output` document that is finally shown to the client.

3.3 Flexibility

In this section, we will evaluate the flexibility of our document construction approach and contrast it to other mechanisms.

A service often needs to display a page presenting a dynamically generated list of data; for instance, the list of results from a search engine. The following JSP example displays twenty entries of an array as options of a `select` field:

```
<select name="choice">
  <option value="1"><%= array[1] %>
  <option value="2"><%= array[2] %>
  ...
  <option value="20"><%= array[20] %>
</select>
```

Clearly, the number of entries must be determined at compile-time and is *hard-wired* into the template. If another number is required, another template must be used.

If the number of options is not known at compile-time, the page must be constructed by one big *generate-all* script element:

```
<select name="choice">
  <%
    for (int i=1; i<=N; i++) {
      out.print("<option value=\"" + i + "\">" + array[i]);
    }
  %>
</select>
```

However, the static HTML for marking up individual entries is now hidden away inside the script element and hard to discern from the programming.

With our mechanism, the same list can, for instance, be generated by two templates and a simple recursive function:

```
html Select = <html><select name="choice"><[options]></select></html>;

html Option = <html><option value=[value]><[option]><[options]></html>;

html genSelect(int n) {
  if (n==0) return Select;
  return genSelect(n-1)<[options = Option<[value=n, option=array[n]]];
}
```

The `Select` template is responsible for rendering the context for the list. It has a gap, *options*, into which the list of entries will be inserted. The `Option` template contains the layout for one entry followed by an *options* gap into which subsequent entries will be plugged. Now, the function `genSelect` may invoked with an arbitrary number; `genSelect(27)` will for instance generate a list with 27 entries.

Note how the HTML markup is completely separated from the program logic. In fact, we can change the layout independent of the program code. If we replace the two templates by:

```
html Select = <html><ul><[options]></ul></html>;

html Option = <html>
  <li><[option]>: <input type="radio" name="choice" value=[value]>
  <[options]>
</html>;
```

the same program instead displays the choices as a bullet list of `radio` buttons. The separation can be further enhanced by placing HTML fragments in individual files and including them through the compile-time lexical inclusion directive, `#include`. As long as the designer and programmer agree on which gaps and fields are in an HTML fragment, they may operate completely independently. To facilitate this, we have added a language construction for dynamic inclusion that makes this *contract* explicit. Any template constant may be followed by `@` and a URL. The semantics is that if an HTML file with the same gaps and fields is found at the end of the URL, then it is used, otherwise the inlined prototype document is used. This enables the programmer to rapidly prototype a service which may then be incrementally improved by the designer.

Note that `<bigwig>` is as general as all other languages for producing XML trees, since it is possible to define for each different element a tiny fragment like:

```
<html><ul type=[type]><[items]></ul></html>
```

that corresponds to a constructor function. The typical use of larger fragments is mostly a convenience for the `<bigwig>` programmer.

Our higher-order construction mechanism overcomes both of the flexibility limitations presented earlier. Not only may documents be constructed top-down and bottom-up, but also as any combination of the two. As we shall see in the next section, the flexibility is gained without sacrificing static safety.

3.4 Safety

The plug and show operations may be misused in a number of ways. A plug operation, `x<[g=y]`, fails:

- if the document held in `x` does not have a *g* gap; or
- if an HTML fragment is plugged into an attribute gap.

The first case could of course be given a sensible semantics; it could evaluate to a copy of `x`. However, we have chosen to interpret this as a program error which ensures that no HTML plugged is “lost”. Plugging HTML elements into an attribute gap would create non-wellformed HTML. A `show-receive` statement fails:

- if a field designated in the receive part is not present in the document shown; or

- if a field is not received or received into a program variable of the wrong type.

This guarantees the interaction correspondence between what is shown and received as mentioned earlier. Regarding the types of values received, we distinguish between *atomic values* and *vector values*. Atomic values are, for instance, produced by a `text` input field or any number of `radio` buttons with the same name. Vector values are produced, for instance, by multiple `checkbox` fields or by a single `select multiple` field permitting any number of items to be selected. If we want to intercept these errors at compile-time, we clearly need to know the names and kinds of gaps and fields present in documents plugged and shown.

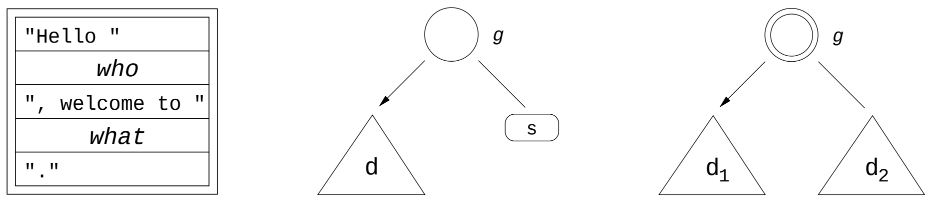
One solution is to explicily declare the exact types of all `html` variables used in the program. However, this means that all gaps and fields along with their individual kinds would have to be described, which may be rather voluminous. Another drawback is that `html` variables would be required to have the same type at all program points. Thus, a document cannot be gradually constructed without the introduction of variables to hold all the temporary documents.

For these reasons, we rely instead on flow-sensitive *type inference* to determine the exact types of all document expressions and variables at all program points. In our experience, this results in a more liberal and useful mechanism.

We employ standard data flow analysis [65] techniques, but with highly specialized lattice structures to represent document types. For every document variable and expression that occurs in the given program, we associate a lattice element that captures the relevant gap and field information and abstracts away everything else. It is possible to define monotone transfer functions which abstractly describe the effect of the program statements.

Given a `<bigwig>` program we now construct a flow graph. This is straightforward since there are no higher-order functions or virtual methods in `<bigwig>`. All language constructs that do not involve documents are abstracted away. This produces a constraint system which we solve using a classical fixed point iteration technique. From this solution, we inspect all plug and show operations and make sure the errors mentioned above do not occur. In case they do, appropriate error messages indicating the causes are generated.

With this approach, the programmer is only restricted by the requirement that at every program point, the template type of an expression must be fixed. In practice, this does not limit the expressibility, rather, it tends to enforce a more comprehensible structure of the programs.



(a) Leaf: `greeting` (b) Node: `strplug(d,`g`,s)` (c) Node: `plug(d`$_1$`,`g`,d`$_2$`)`

Figure 3.2: Constituents of the DynDocDag representation.

This was implemented in `<bigwig>` as a monovariant and interprocedural data flow analysis. However, extensive evaluation exposed a recurring annoyance. Often, gaps were only plugged along one branch and not the other, yielding errors at such points of confluence. Consequently, programmers needed to *explicitly* plug in the empty string along the other branch. Since lots of code was dedicated to this, we decided instead to automate this process. It was solved by carefully placing *gap absence* above *gap presence* in the lattice, so that a *least-upper-bound* of the two kinds at confluence points yielded absence of a gap. To maintain a consistent runtime representation, the solution inspection was augmented to instrument the code, by inserting the missing plug statements. We believe this *implicit plugging* increases the usability of our document construction mechanism.

Another important safety aspect is to ensure that only valid HTML documents are ever shown to clients. This task will be covered extensively in Chapter 4.

3.5 Efficiency

Having devised the document construction mechanism, we need an efficient representation to handle documents at runtime. Representing documents naively as complete parse trees, would cause the space complexity of a document to be proportional to its printed size. Also, the time complexity of the plug operation would be linear in the lexical sizes of the documents involved. The paper [72] describes a switchboard data structure that shares all template constants involved and supports the plug in constant time. However, that data structure does not support multiple gaps with the same name and cannot be generalized to do so without compromising the constant plug time bound.

The following will present an even more efficient data structure, DynDocDag, that overcomes this limitation.

A dynamic document is at runtime represented as a binary directed acyclic graph. The leaves are either HTML or string constants that have been plugged into the document and the nodes represent pluggings that have constructed the document. The data structure supports four operations: constructing constant documents, `constant(c)`; string plugging, `strplug(d,`g`,s)`; document plugging, `plug(d`$_1$`,`g`,d`$_2$`)`; and showing documents, `show(d)`.

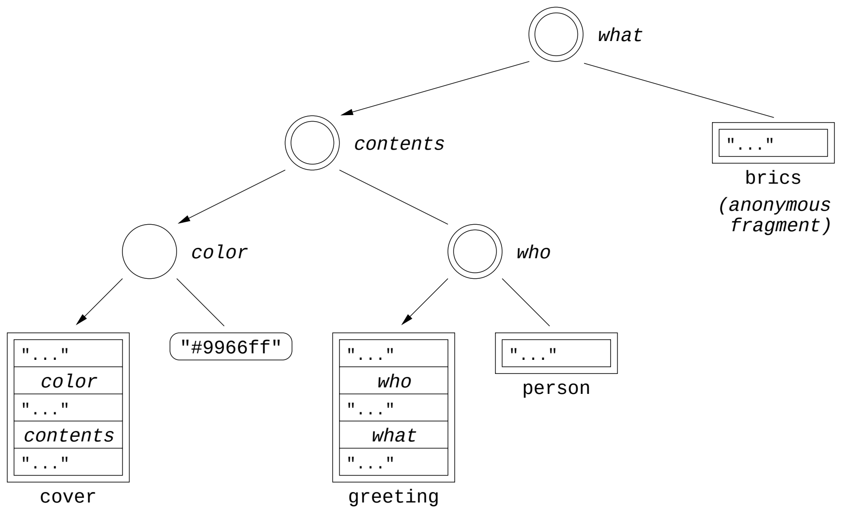


Figure 3.3: DynDocDag representation of the document shown in the BRICS example.

A constant template is represented as an ordered sequence of its text and gap constituents. For instance, the `greeting` template from the "Welcome to BRICS" example is represented as displayed in Figure 3.2(a); it has three text entries with the two gaps between them. A constant template is *shared* among the documents it has been plugged into and thus only represented once in memory. This causes the data structure to be a DAG in general and not a tree.

The string plug operation, `strplug`, combines a DAG and a constant string by adding a new string plug root node with the name of the gap, as illustrated in Figure 3.2(b). Analogously, the `plug` operation combines two DAGs as shown in Figure 3.2(c). For both operations, the left branch is the document containing the gap being plugged and the right branch is the value being plugged into the gap. Thus, the data structure merely records plug operations and defers the actual document construction to subsequent `show` operations.

Figure 3.3 shows the representation of the document constructed in the "Welcome to BRICS" example.

The `show` operation linearizes a document DAG by recursively traversing the DAG data structure. The `constant(c)`, `strplug(d,`g`,s)`, `plug(`$\mathtt{d}_1$`,`g`,`$\mathtt{d}_2$`)`, and `show(d)` operations have optimal complexities, $O(1)$, $O(1)$, $O(1)$, and $O(|\mathtt{d}|)$[1], respectively, where $|\mathtt{d}|$ is the lexical size of the `d` document.

Even though the asymptotic complexities of the plug and show operations are the same as that of the old switchboard representation, the new DynDocDag is considerably faster. Plugging a simple document into itself 20.000 times in a `C` implementation was 7 times faster with the new representation and linearization of this document was 3 times faster.

Note that for some documents, the representation is exponentially more succinct than the expanded document. This is for instance the case with the

[1]Actually, this bound assumes there are no textless documents with gaps.

following function:

```
html list = <html><ul><li><[gap]><li><[gap]></ul></html>;

html tree(int n) {
  if (n==0) return <html>foo</html>;
  return list<[gap=tree(n-1)];
}
```

which given n, in $O(n)$ time and space will produce a document of lexical size $O(2^n)$.

In Chapter 5 we will show how to push the show complexity even further to sub-linear time complexities by exploiting the browser's standard caching mechanism.

3.6 Other Related Work

Guide [57] provides a flat template mechanism much like that of MAWL templates. The templates may contain named gaps and fields which are implicitly plugged with and received into variables with the same name in the associated context.

In WASH/CGI [85], a notion of *monad transformers* permits higher-order and first-class document construction. It is embedded in a GPL (Haskell) and thus requires intricate details of this underlying host language and monads. In particular, HTML construction must be conducted through explicit invocation of Haskell constructor libraries which do not look and feel like HTML; any library misuse is signalled as Haskell errors.

Input field handlers are bound to input fields and thus permit the interaction correspondence to be type checked. The relationship between input handlers and HTML permits modular construction of new input widgets. Similar functionality could conceivably be achieved with our documents by adding receive code to collate the input fields of a template into externally visible abstract input fields which could then be received.

The Curry [40] library is also capable of typing the interaction correspondence by using logical variables to tie input fields with submit handler code.

XDuce [43, 44] is a statically typed domain specific language for XML *processing*. It has a notion of first-order XML documents typed with *regular expression types* which correspond to schemas. Values in the program are statically typed using explicit programmer annotations. The flexibility of this approach is made practically useful by a subtyping relation which is based directly on tree automata. It also provides a typed document deconstruction mechanism based on regular expression pattern matching.

JWIG [26] is the successor of the `<bigwig>` language. It is essentially Java, extended with the sessions, dynamic documents, and form field validation concepts of `<bigwig>`.

The main differences pertaining to document construction are that the interaction correspondence analysis is run on summary graphs which are introduced in the next chapter instead of using special gap and field lattices. In JWIG, gaps are never implicitly plugged, just not pluggable if not present on all branches. Also, the receive part is detached from the show statement meaning that unused input values do not have to be received.

3.7 Conclusion

We have presented a domain specific language capable of dynamically generating XML documents in a flexible, safe, and efficient way that solves all four problems mentioned in the introduction.

Chapter 4

Static Validation of Dynamically Generated XML

4.1 Introduction

In this chapter we will look at how to ensure that clients are only presented with *valid* HTML documents in the sense that they conform to the official DTD for HTML 4.01, or rather XHTML 1.0 [67]. For static HTML documents this is easy; they can readily be validated by tools made available by W3C and others. For documents dynamically generated by scripts a frequently employed strategy is to validate them after they have been produced at runtime. However, this is an incomplete and costly process which does not provide any static guarantees about the behavior of the script.

In this chapter we will show how to analyse our document construction mechanism to statically guarantee that only valid HTML documents are ever presented to clients.

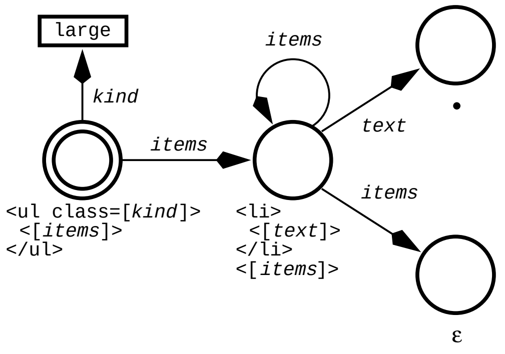
In short, our approach is to first conservatively approximate the possibly infinite set of XML documents that may be constructed at `show` statements, capture the infinite set of *valid* XML documents through a schema formalism, and finally decide validation as the inclusion of these two sets of XML documents.

4.2 Summary Graph Analysis

Like in the previous chapter, we will employ standard data flow-analysis techniques to collect information about documents. This time, our lattice will consist of *summary graphs* that approximates the set of HTML documents that a given document expression may evaluate to. This structure essentially records all plug operations involved in the construction of a document.

More precisely, given a set, N, of template constants, a set, G, of gap names, and a set, C, of constant strings occuring in the program, a summary graph has three constituents (R, E, α). The first constituent, $R \subseteq N$, is a *root set* that designates the possible outermost templates in the document. The second, $E \subseteq N \times G \times N$, is an *edge set* that contains an edge from template n to m labelled g, written (n, g, m), if template m has been plugged into the g gap of template n. Finally, $\alpha : G \times N \rightarrow S$, where $S = 2^C \cup \{\bullet\}$, is a *labelling function* that, for each gap of each template contains either the set of constant strings have been plugged into the gap or "$\bullet$" if it may contain strings whose values cannot be determined at compile-time.

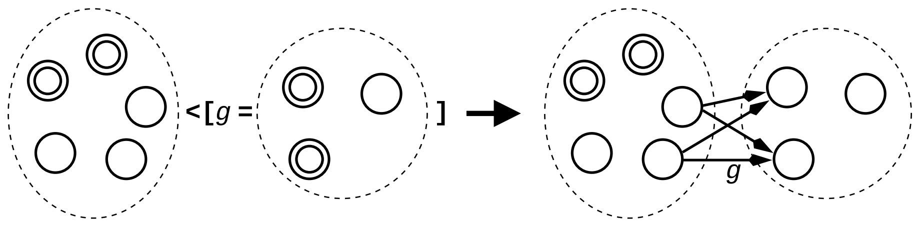
As an example, consider the following summary graph consisting of one root template node, four plug edges, and a single attribute labeling:



Template nodes, root nodes, and attribute labels are drawn as circles, double circles, and boxes, respectively. The "ϵ" node models the empty template.

Each summary graph G defines a possibly infinite set of XML documents, denoted $\mathcal{L}(G)$. Intuitively, this set is obtained by *unfolding* the graph from each root while performing all possible pluggings enabled by the edges and the labeling function. The language of the summary graph depicted above is the set of all `ul` lists of class `large` with one or more character data items. Summary graphs turns out to provide an ideal abstraction level for verifying HTML validity.

It is possible to model the document and string plug operations with good precision using transfer functions. The plug transfer function takes two summary graphs and a gap name. The second summary graph is plugged into the first by adding edges from all relevant template gaps of the first to the roots of the second. Ignoring the internal edges of each of the two summary graphs, here depicted as disjoint, the plug operation can be illustrated as follows:



Similarly, a string plug transfer function models the effect of plugging in a string.

However, in order to achieve sufficient precision of this analysis, two preliminary analyses are required. One for tracking string constants, and one, called a *gap track analysis*, for tracking the origins of gaps. The latter tells us for each template variable and gap name, which constant templates containing such a gap can flow into that variable at any given program point. This helps cut down the number of new edges introduced by the summary graph plug operation. Clearly, all these analyses are highly specialized for the domain of dynamic document construction and for `<bigwig>`'s higher-order template mechanism, but they all fit into the standard data-flow analysis frameworks. For more details we refer to [17].

4.3 An Abstract DTD for XHTML

Once we have the summary graphs for all the `show` statements, we need to verify that the sets of documents they define all are valid HTML according to W3C's official definition. To simplify the process we reformulate the notion of Document Type Definition (DTD) as a simpler and more convenient formalism that we call *abstract DTD*. An abstract DTD consists of a number of *element declarations* whereof one is designated as the root. An element declaration defines the requirements for a particular type of elements. Each declaration consists of an element name, a set of names of attributes and subelements that may occur, and a boolean expression constraining the element type instances with respect to their attribute values and contents. The official DTD for HTML is easily rewritten into our abstract DTD notation. In fact, the abstract DTD version captures more validity requirements than those expressible by standard DTDs and merely appear as comments in the HTML DTD. As a technicality we actually work with XHTML 1.0 which is an XML reformulation of HTML 4.01. There are no conceptual differences, except that the XML version provides a cleaner tree view of documents for the analysis.

4.4 Validation

Given a summary graph and an abstract DTD description of HTML, validity can be checked by a recursive traversal of the summary graph starting at the roots. We memoize intermediate results to ensure termination since the summary graphs may contain loops. If no violations are encountered, the summary graph is valid. Since all validity properties are local to single elements and their contents, we are able to produce precise error messages in case of violations. Analysis soundness is ensured by the following property: if all summary graphs corresponding to `show` expressions are verified to be valid with respect to the abstract DTD, then all concrete documents are guaranteed to be valid HTML.

4.5 Experiments

The program analyses described here all have high worst-case complexities because of the complex lattices. Nevertheless, our implementations and experiments show that they work well in practice, even for large intricate programs.

The validation analysis has been fully implemented as part of the `<bigwig>` system. It has then been applied to all available benchmarks, some of which are shown in the following table:

Name	**Lines**	**Templates**	**Size**	**Shows**	**Sec**
`chat`	65	3	(0,5)	2	0.1
`guess`	75	6	(0,3)	6	0.1
`calendar`	77	5	(8,6)	2	0.1
`xbiff`	561	18	(4,12)	15	0.1
`webboard`	1,132	37	(34,18)	25	0.6
`cdshop`	1,709	36	(6,23)	25	0.5
`jaoo`	1,941	73	(49,14)	17	2.4
`bachelor`	2,535	137	(146,64)	15	8.2
`courses`	4,465	57	(50,45)	17	1.3
`eatcs`	5,345	133	(35,18)	114	6.7

The entries for each benchmark are its name, the lines of code derived from a pretty print of the source with all macros expanded, the number of templates, the size $(|E|, |\alpha|)$ of the largest summary graph, the number of program points with `show` statements, and the analysis time in seconds (on an 800 MHz Pentium III Linux PC).

The analysis found numerous validation errors in all benchmarks, which could then be fixed to yield flawless services. No false errors were reported. As seen in the table above, the enhanced compiler remains efficient and practical.

Error Diagnostics

The `<bigwig>` compiler provides detailed diagnostic messages in case of validation errors. For the flawed example:

```
service {
  html cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolo=[color]>
      <table><[contents]></table>
    </body>
  </html>;

  html greeting = <html>
    <td>Hello <[who]>,<br clear=[clear]>
        welcome to <[what]>.
    </td>
  </html>;

  html person = <html>
    <i>Stranger</i>
  </html>;

  session welcome() {
    html h;
    h = cover<[color="#9966ff",
               contents=greeting<[who=person],
               clear="righ"];
    show h<[what=<html><b>BRICS</b></html>];
25   }
26 }
```

the compiler generates the following messages for the single **show** statement:

```
--- brics.wig:24: HTML validation:
brics.wig:4:
  warning: illegal attribute 'bgcolo' in 'body'
  template: <body bgcolo=[color]><form>...</form></body>

brics.wig:5:
  warning: possible illegal subelement 'td' of 'table'
  template: <table><[contents]></table>
  contents: td
  plugs: contents:brics.wig:22

brics.wig:10:
  warning: possible element constraint violation at 'br'
  template: <br clear=[clear]/>
  constraint: value(clear,left,all,right,clear,none)
  plugs: clear:brics.wig:23
```

At each error message, a line number of an XML element is printed together with an abbreviated form of the involved template, the names of the root elements of each template that can be plugged into the gaps, the constraint being violated, and the line numbers of the involved plug operations. Such reasonably precise error diagnostics is clearly useful for debugging.

4.6 Related Work

Since the documents of MAWL [55, 3, 4] and Guide [57] are restricted to templates that are only parameterizable with character data, they may be prevalidated.

The paper [84] shows how to achieve validity by encoding a DTD as instance classes in an extension of Haskell's type system. However, the author reports that the encodings are too restrictive to be practically useful for generating parameterized documents unless the validity requirements for attributes are relaxed. In addition to documents having to be composed in an *element-transforming style*, validation errors are reported as Haskell instance class type errors which may be hard to "decode".

As previously mentioned, all XDuce [43, 44] values are statically typed with *regular expression types* which are essentially equivalent to DTD's. In its present form, XDuce is incapable of coping with attributes. However, preliminary attempts to integrate attributes have been made [42], but it is unclear how this is to proceed. Apart from attributes, XDuce achieves validation, but relies on explicit programmer annotations to do so.

JWIG [26] has extended the validation presented here from *abstract DTD's* to the more powerful schema language, DSD2 [51].

4.7 Conclusion

We have combined a data-flow analysis with a generalized validation algorithm to enable the `<bigwig>` compiler to guarantee that all HTML or XHTML documents shown to the client are valid according to the official DTD. The analysis is efficient and does not generate many spurious error messages in practice. Furthermore, it provides precise error diagnostics in case a given program fails to verify.

Since our algorithm is parameterized with an abstract DTD, our technique generalizes in a straightforward manner to arbitrary XML languages that can be described by DTDs. In fact, we can even handle more expressive grammatical formalisms. The analysis has proved to be feasible for programs of realistic sizes. All this lends further support to the unique design of dynamic documents in the `<bigwig>` language.

Chapter 5

Caching of Dynamically Generated XML

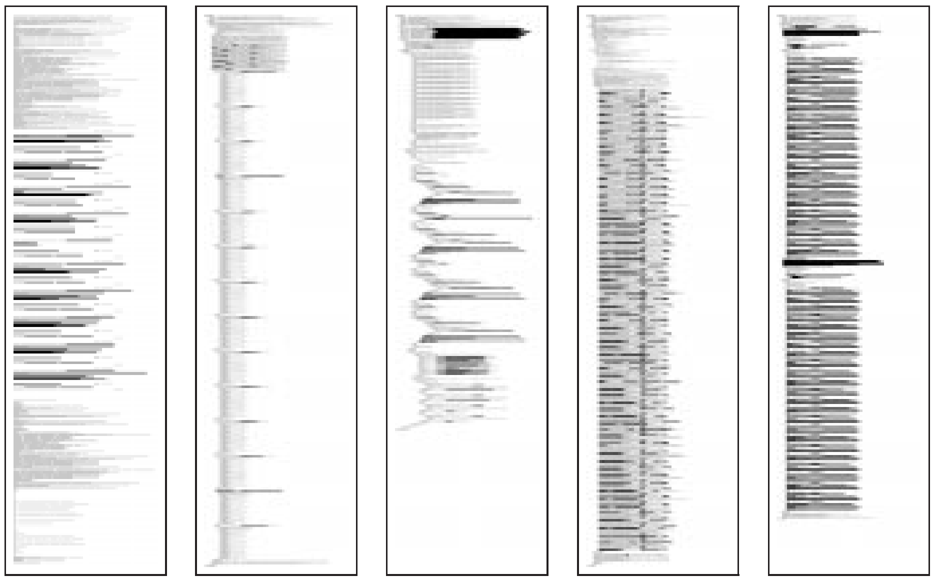
5.1 Introduction

Caching documents on the client-side is an important technique for saving bandwidth, time, and clock-cycles. The HTTP protocol provides explitcit support for this by associating an "expiration time" with all documents sent from the server to the client. A document that never or rarely changes may then be associated an appropriate future expiration time so that browsers and proxy servers may avoid reloading it before that time. However, this mechanism is clearly not applicable to dynamically generated documents that change on every request. For such documents, the expiration must always be set to "now", voiding the benefits of caching.

Though caching does not work for *whole* dynamically generated documents, most Web services construct HTML documents using some sort of constant parts that ideally ought to be cached, as also observed in [31, 97]. In Figure 5.1, we show a condensed view of five typical HTML pages generated by different Web services using the document construction mechanism described in Chapter 3. Each column depicts the dynamically generated raw HTML text output produced from interaction with each of our five benchmark Web services. Each non-space character has been colored either grey or black. The grey sections are characters that originate from a large number of small, constant HTML templates in the source code; the black sections are dynamically computed strings of character data, specific to the particular interaction. The templates appear to constitute a significant part of generated documents.

Experiments have shown that many of our templates tend to occur again and again in documents shown to a client across the lifetime of a service, either because they occur 1) many times in the same document, 2) in many different documents, or 3) simply in documents that are shown many times.

Since the templates account for a large part and reoccur there is potentially much to gain if they could be cached on the client. We will now show how to exploit our document generation mechanism to do this.



(a) `lycos` (b) `bachelor` (c) `jaoo` (d) `webboard` (e) `dmodlog`

Figure 5.1: Benchmark services: cachable (grey) vs. dynamic (black) parts.

5.2 Our Solution

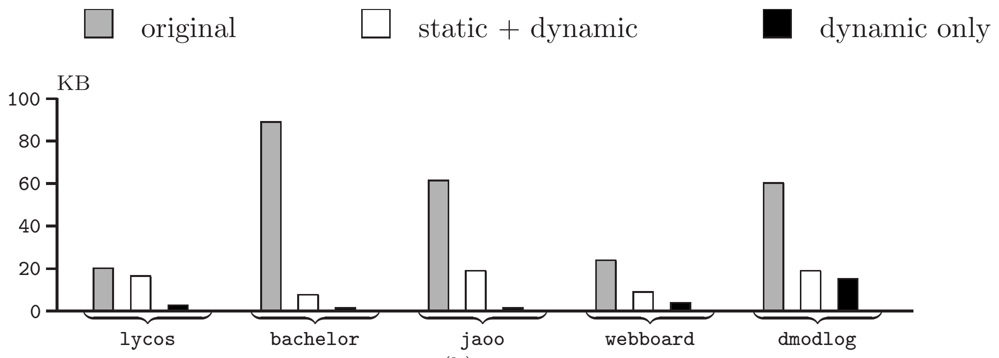
The DynDocDag representation described in Section 3.5 has a useful property: it explicitly maintains a separation of the *constant templates* occurring in a document, the *strings* that are plugged into the document, and the *structure* describing how to assemble the document. In Figure 3.3, these constituents are depicted as framed rectangles, oval rectangles, and circles, respectively.

The templates are inherently static. The strings and structure of a document, however, are typically customized for an individual interaction and thus change with each document.

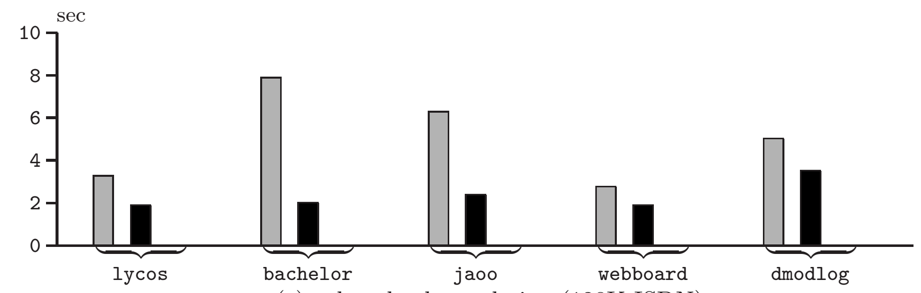
The solution is to move the unfolding of the DynDocDag data structure from the server to the client. Instead of transmitting the unfolded HTML document, the server will now transmit only a compact representation of the dynamic parts of the DynDocDag data structure along with some generic JavaScript code capable of reconstructing the document on the client. The templates are not present in the document transmitted, but each placed in its own JavaScript file on the server and merely referenced by a JavaScript include directive in the file transmitted. This way, each JavaScript template file can be cached by the browser just as any other file. In fact, the generic JavaScript unfolding code can also be placed in its own file and cached.

Consequently, only the dynamic string and structure constituents are transmitted; the browser's standard caching mechanism will ensure that templates already seen are not reloaded.

Since the templates are statically known at compile-time, the compiler can



(b) size



(c) download+rendering (128K ISDN)

Figure 5.2: Experiments with the template representation.

enumerate the templates and for each of them generate a file with appropriate JavaScript code. By postfixing template filenames with version numbers, caching can be enabled across recompilations where only certain templates have been modified.

We could have chosen Java instead of JavaScript, but JavaScript is more lightweight and is sufficient for our purposes. Alternatively, similar effects could be obtained using browser plug-ins or proxies, but implementation and installation would become more difficult.

Our approach could be adapted to languages such as ASP, PHP, JSP, but the cachable text and markup we have in one structured template would essentially have to be cut into the strings between the gaps. These pieces are likely to be smaller and unstructured yielding a bigger overhead and less opportunities for reuse.

5.3 Evaluation

Figure 5.2 recounts the effect of applying our caching technique to the five Web service benchmarks mentioned earlier.

In Figure 5.2(b) we show the sizes of the data transmitted to the client. The grey columns show the original document sizes, ranging between 20 and 90 KB. The white columns show the sizes of the total data that is transmitted using our technique, none of which exceeds 20 KB. Of ultimate interest is the black column which shows the *asymptotic* sizes of the transmitted data which are reached when all templates have been cached by the client. In this case, we see reductions of factors between 4 and 37 compared to the original document size. When employing our technique, the amount of data downloaded is anywhere between what indicated by the white and black columns, depending on how many of the templates has been cached. In any case, our technique substantially reduces the number of bytes transmitted from the server to the client.

The HTTP 1.1 protocol [37] introduces automatic compression using general-purpose algorithms, such as `gzip`. Of course, adding compression drastically reduces the benefits of our caching technique. However, we still see asymptotic reduction factors between 1.3 and 2.9 suggesting that our approach remains worthwhile even in these circumstances. Clearly, there are documents for which the asymptotic reduction factors will be arbitrarily large, since large constant text fragments count for zero on our side of the scales while `gzip` can only compress them to a certain size. Thus, compression is essentially orthogonal to our approach.

In cutting down the network traffic, we of course seize client clock cycles for the unfolding. However, in a context of fast client machines and comparatively slow networks, this is a sensible tradeoff. In Figure 5.2(c) we quantify the end-to-end latency for our technique. The total download and rendering times for the five services are shown for both the standard documents and our cached versions. The client is Internet Explorer 5 running on an 800 MHz Pentium III Windows PC connected to the server via a 128K ISDN modem. These are still realistic configurations, since by August 2000 the vast majority of Internet subscribers used dial-up connections [46] and this situation will not change significantly within the next couple of years [64]. The times are averaged over several downloads (plus renderings) with browser caching disabled. The download and rendering times reduce by factors between 1.4 and 3.9. Even the `dmodlog` benchmark which presents lots of dynamically generated data benefits in this setup. For higher bandwidth dimensions, the results will of course be less impressive.

5.4 Related Work

Caching of dynamic contents has received increasing attention the last years since it became evident that traditional caching techniques were becoming insufficient.

Most existing techniques labeled "dynamic document caching" are server-based, for instance [69, 25, 47, 100]. The primary goal for server-based caching techniques is not to lower the network load or end-to-end latency as we aim for, but to relieve the server by memoizing the generated documents in order to avoid redundant computations. Such techniques are orthogonal to the one we propose. Other techniques are proxy-based, e.g. [23, 77], and require the installation of special proxy servers. Our technique and the HPP language [31] are client-based and neither require intrusive modifications to existing protocols.

The HPP language [31] is closely related to our approach. Both are based on the observation that dynamically constructed documents usually contain common constant fragments. HPP is an HTML extension which allows an explicit separation between static and dynamic parts of a dynamically generated document. The static parts of a document are collected in a *template* file while the dynamic parameters are in a separate *binding* file. The template file can contain simple instructions, akin to embedded scripting languages such as ASP, PHP, or JSP, specifying how to assemble the complete document. According to [31], this assembly and the caching of the templates can be done either using cache proxies or in the browser with Java applets or plug-ins, but it should be possible to use JavaScript instead, as we do.

An essential difference between HPP and our approach is that the HPP solution is not integrated with the programming language used to make the Web service. With some work it should be possible to combine HPP with popular embedded scripting languages, but the effort of explicitly programming the document construction remains. Our approach is based on the source language, meaning that all caching specifications are automatically extracted from the Web service source code by the compiler and the programmer is not required to be aware of caching aspects. Regarding cachability, HPP has the advantage that the instructions describing the structure of the resulting document are located in the template file which is cached, while in our solution the equivalent information is in the dynamic file. However, in HPP the constant fragments constituting a document are collected in a single template. This means that HTML fragments that are common to different document templates cannot be reused by the cache. Our solution is more fine-grained since it caches the individual fragments separately. Also, HPP templates are highly specialized and hence more difficult to modify and reuse for the programmer. Being fully automatic, our approach guarantees cache soundness. Analogously to optimizing compilers, we claim that the `<bigwig>` compiler generates caching code that is competitive to what a human HPP programmer could achieve. This claim is substantiated by our `lycos` benchmark which is equivalent to one presented for HPP [31], except that our reconstruction is of course in `<bigwig>`. It is seen that the size of our residual dynamic data (from 20,183 to 3,344 bytes) is virtually identical to that obtained by HPP (from 18,000 to 3,250 bytes). However, in that solution all caching aspects are hand-coded with the benefit of human insight, while ours is automatically generated by the `<bigwig>` compiler. The other four benchmarks construct more complicated documents and would be more challenging for HPP.

5.5 Conclusion

With our approach, the programmer need not be aware of caching issues since the decomposition of pages into cachable and dynamic parts is performed automatically by the compiler. The resulting caching policy is guaranteed to be sound, and experiments show that it results in significantly smaller transmissions and reduced latency. Our technique is non-intrusive and requires no extensions to existing protocols, clients, servers, or proxies.

As a result, we obtain a simple and practically useful technique for saving network bandwidth and reviving the cache mechanism present in all modern Web browsers in the context of dynamically generated Web pages.

Chapter 6

Form Field Validation

6.1 Introduction

A considerable effort in Web programming is expended on making sure the data supplied by the client in a form input field has the right format. A field might for instance expect a valid number, date, or email address to be entered in a certain way.

This is often achieved through *server-side input validation*. When the page containing the input fields is submitted, the program on the server determines whether the entered data is of the required form. If this is not the case, the program outputs a page containing appropriate error messages along with the erroneous input fields, allowing the client to correct them. This process is repeated until all input fields contain valid data. Although widely used, the approach has some considerable drawbacks

- it takes time;
- it causes excess network traffic; and
- it requires explicit programming.

Note that these drawbacks affect all parties involved. The client is clearly annoyed by the extra time incurred by the round-trip to the server for validation, the server by the extra network traffic and "wasted" cycles, and the programmer by having to explicitly wrap the showing of such documents in loops that retransmit documents along with appropriate error messages until the input validates. Also, adding extra control structures clutter up the main logic of the service with validation code.

The first two drawbacks are solved by moving the validation from the server to the client, yielding *client-side input validation*. The actual validation is then undertaken by a client-side scripting language, typically JavaScript.

The move from server-side to client-side also opens for another important benefit, namely the possibility of performing the validation *incrementally*. The client no longer needs to click the submit button before getting the validation report. This allows errors to be be signalled as they occur, which clearly eases the task of correctly filling out the form. Also, the browser features made available in the scripting language may help provide more sophisticated interactions with the client, such as pop up error and help messages and coloring of erroneous input fields.

However, writing JavaScript input validators that at the same time capture all validity requirements and also signal errors appropriately is a tedious and error-prone task and is further complicated by diverging browser implementations. In fact, whole Web sites are dedicated to explaining how JavaScript implementations differ in browsers[1].

[although many libs exist [2], they must be used in the context of a GPL]

Since JavaScript may be unsupported or disabled by the client, the server must always perform a second validation. Thus, the same code must essentially be written both in the client and server scripting languages that may be very different in nature.

6.2 PowerForms

To address these issues we have designed a language, PowerForms, targeted uniquely at the domain of input validation.

We allow the service programmer to define formats and attach them to textual input fields. Submission of form input to the server is prohibited while data entered in the input fields does not comply with the attached formats. Thus, clients are only allowed to continue a session when all input fields contain appropriate data.

The formats are specified as standard regular expressions enhanced with intersection, complement, and integer intervals for convenience. There are several motivations for choosing regular expressions over other specification formalisms, such as context-free grammars or Turing-complete languages. Regular expressions are simple, well known, and widely used for other text pattern-matching purposes, as for instance in Perl.

Also, regular expressions are inherently *declarative* and thus abstract away all operational details making the validation easier to read, write, and modify. Comparatively, operational formalisms, such as JavaScript, force programmers to deal with details of *how* fields and contents are validated and in what order. Since programming in the operational sense is not required, input validation is available to a wider audience.

Finally, it can be efficiently decided whether a string is in the language defined by a regular expression through the use of deterministic finite automata, DFAs.

It is our experience that regular expressions are sufficiently expressive to capture most common validation requirements, such as validating dates, email addresses, and zip-codes. Anything requiring expressiveness beyond regularity is deferred to the server-side, but this is rarely needed.

[1] See e.g. `http://www.webdevelopersjournal.com/articles/javascript_limitations.html` or `http://www.xs4all.nl/ ppk/js/version5.html`.

[2] http://developer.netscape.com/docs/examples/javascript/formval/overview.html

When a document is shown, all its regular expressions are compiled into minimized DFAs and the HTML is instrumented with JavaScript code to incrementally run the automata on the data entered in their associated input fields. These automata are also used on the server side to double check the submitted data upon reception. The compilation only generates code within the subset of JavaScript that is known to work on all common browser implementations.

Regarding efficiency, all automata are remembered to avoid trivial recompilation and placed in individual JavaScript files so that they may be cached by the browser. Also, the `<bigwig>` compiler will statically compile all regular expressions available at compile-time; only dynamically generated regular expressions are subjected to dynamic compilation.

To provide continuous feedback to the client about the state of the validation, we visualize the states of all automata using images displayed next to all textual input fields. These images are then dynamically changed to always reflect the state of an automaton. By default, the compiler uses "traffic light" icons, displaying either red, yellow, or green light corresponding to whether an automaton is in a crash, reject, or accept state when run on its input. Thus, red means not prefix of valid input; yellow, strict prefix of valid input; and green, valid input. Input fields may be instructed to use other icons or means for visualization or even to auto-complete when there is only one possible suffix yielding valid input.

An Example

The definition of formats is syntactically disjoint from the form itself. This allows a modular development in that validation can be added to an input field in an existing HTML form without knowing anything but its name. Consider for instance an HTML document with an input field, ***address***, expecting a valid email address:

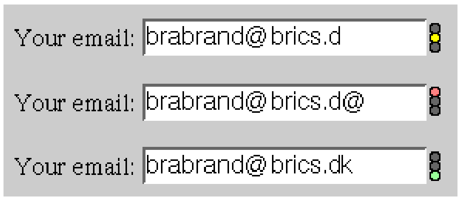
```
<html> .. Your email: <input type="text" name="address" size="25"/> .. </html>
```

Email addresses are easily captured by a regular expression. Valid emails could for instance be defined as follows, assuming ***word*** is appropriately defined:

```
<regexp id="email">
  <regexp idref="word"/>
  <const value="@"/>
  <plus>
    <regexp idref="word"/>
    <const value="."/>
  </plus>
  <repeat low="2" high="3">
    <range low="a" high="z"/>
  </repeat>
</regexp>
```

or alternatively in a more compact Perl-style syntax:

```
<regexp id="email" exp="<word>@(<word>\.)+[a-z]{2,3}"/>
```

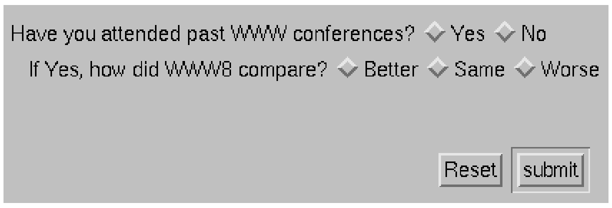


Your email: brabrand@brics.d

Your email: brabrand@brics.d@

Your email: brabrand@brics.dk

Figure 6.1: Checking email addresses.



Have you attended past WWW conferences? ◇ Yes ◇ No

If Yes, how did WWW8 compare? ◇ Better ◇ Same ◇ Worse

Figure 6.2: Conference questionnaire.

An email is here defined as the concatenation of a word, a "`@`" character, one or more words followed by a dot, and two alphanumeric characters. This format can then be independently added to the HTML page by the following definition:

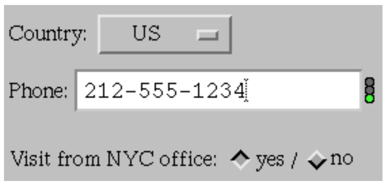
```
<format field="address" help="Enter email address" error="Illegal email">
  <regexp idref="email"/>
</format>
```

The `field` attribute refers to the input field to which the regular expression is to be bound. While the input field has focus, the *help* string appears in the status line of the browser. If the client attempts to submit the form with invalid data in this field, then the *error* text appears in an alert box. Initially, the field has a yellow light. This status persists, as seen in Figure 6.1, while we enter the text "`brabrand@brics.d`" which is a legal prefix of an email address. Entering another "`@`" yields a red light. Deleting this character and entering `k` will finally give a legal value and a green light.

6.3 Field Interdependency

Many forms contain fields whose values are constrained by selections made or text entered in other fields. Figure 6.2 exhibits a simple questionnaire from a conference, in which participants were invited to state whether they have attended past conferences and if so, how this one compared. The second question clearly depends on the first, since it may only be answered if the first answer was positive. Conversely, an answer to the second question may be required if the first answer was "Yes".

Such interdependencies are almost always handled on the server, even if the rest of the validation is addressed on the client-side. The reason is presumably that interdependencies require even more delicate JavaScript code.


Country: US

Phone: 212-555-1234

Visit from NYC office: yes / no

Figure 6.3: Collecting customer information.

To address these issues, we have made two extensions. Firstly, we permit formats to be associated with all kinds of input fields, not just textual ones. Individual `checkbox` and `radio` buttons are automatically depressed and cannot be checked if their values are not in the language of the regular expression. For `select` fields, illegal options are automatically deselected and filtered from the menu.

Secondly, formats are extended to describe *boolean decision trees* whose conditions probe the values of other fields and whose leaves are simple regular expression formats. In addition to conjunction, disjunction, and negation, two basic predicates, `equal` and `match`, exist for specifying boolean expressions. The `match` predicate takes a field name and a regular expression and decides whether the value of the designated input field is in the language of the regular expression. The `equal` predicate is a shorthand for comparing the value to a constant string.

As an example, consider the form displayed in Figure 6.3 where customers select their country, write their phone number, and check whether or not the want a visit from the New York City office. Since this last option is only available to customers living in New York City, it is constrained by the following format:

```
<format field="visit">
  <if>
    <and>
      <equal name="country" value="US"/>
      <match name="phone"><regexp exp="(212|347|646|718|917).*"/></match>
    </and>
    <then><regexp exp="yes|no"/></then>
    <else><regexp exp="no"/></else>
  </if>
</format>
```

Both “`yes`” and “`no`” are acceptable values for the `visit` field if the “US” option is selected in the *`country`* field and the *`phone`* text field contains a New York City area-code prefix. Otherwise, “`no`” is the only value accepted.

Since the evaluation of a format may produce side-effects in that selections may be unmade, the order in which formats are evaluated clearly matters. We have chosen to process the formats in the sequence they appear in a document because this typically coincides with the order in which the client is supposed to consider them. All formats are processed repeatedly until a fixed-point is reached. Since buttons can only be released, this iteration is guaranteed to terminate. It is our experience that in practise, the evaluation order does not matter and the fixed point is reached in one or two iterations.

6.4 Related Work

ColdFusion [21] provides direct support for server-side validation. However, the validation produces an error report involving the internal names of input fields which are unknown to the client. Also, the required corrections must be remembered when the erroneous form is redisplayed.

The XHTML-FML language [73] provides client-side input validation by adding an attribute to textual input fields. However, this attribute is restricted to a collection of predefined input validation types and there is no support for field inderdependency. The validation is compiled into JavaScript and their solution is non-intrusive in that it does not require installation of special software on the client.

The *extensible form description language*, XFDL[3], is a more elaborate language that deals with the whole lifecycle of a form including workflow. It provides simple interdependeny and a rigid mechanism for defining new formats which is not flexible enough to permit the definition of valid emails as in the New York City example.

XForms [32] is a proposal from W3C that separates the data and presentation of forms. Form data is represented and returned to the server as XML. Validation of this XML data is based on XML Schema [88] and interdependency is achieved using XPath [27] for referencing other parts of the data on a form. However, it is not capable of handling the dependency in the New York City example without involving operational programming. The main problem with XForms is that no complete implementation exists and that it requires an XForms processor on the client.

Both XForms and XFDL have a really useful feature in that fields may be hidden and visualized incrementally as they are required.

The paper [75] is similar to our approach in that it translates a language into client-side validation based on JavaScript and server-side revalidation code. However, validation is not performed incrementally on the client and specification requires explicit programming in a general purpose functional language.

6.5 Conclusion

PowerForms provides incremental and interdependent validation in a declarative way that does not require programming skills. Furthermore, it is modular in the sense that validation can be added to an input field in an existing HTML form without knowing anything but its name. The validation markup being completely separate from the form markup allows the layout of a form to be redesigned at any time in any HTML editor. PowerForms is fully implemented as part of the `<bigwig>` language and is also available as a stand-alone tool[4].

[3]available from `http://www.pureedge.com/xfdl/`

[4]available from `http://www.brics.dk/bigwig/powerforms/`. A Java implementation is also available from `http://www.brics.dk/~ricky/powerforms/` which is integrated in JWIG.

Chapter 7

Concurrency Control

7.1 Introduction

As services have several session threads, there is a need for synchronization and other concurrency control to discipline the concurrent behavior of the active threads. A simple case is to control access to the shared variables using mutex regions or the readers/writers protocol. Another issue is enforcement of priorities between different session kinds, such that a management session may block other sessions from running. Another example is event handling, where a session thread may wait for certain events to be caused by other threads.

We deal with all of these scenarios in a uniform manner based on a central controller process in the runtime system, which is general enough to enforce a wide range of safety properties [71]. The support for concurrency control in the previously mentioned Web languages is limited to more traditional solutions, such as file locking, monitor regions, or synchronized methods.

7.2 Our Solution

A `<bigwig>` service has an associated set of *event labels*. During execution, a session thread may request permission from the controller to pass a specific event checkpoint. Until such permission is granted, the session thread is suspended. The policy of the controller must be programmed to maintain the appropriate global invariants for the entire service. Clearly, this calls for a domain-specific sub-language. We have chosen a succinct, well-known, and very general formalism, temporal logic. In particular, we use a variation of monadic second-order logic [87]. A formula describes a set of strings of event labels, and the associated semantics is that the trace of all event labels being passed by all threads must belong to that set. To guide the controller, the `<bigwig>` compiler uses the MONA tool [50] to translate the given formula into a minimal deterministic finite-state automaton that is used by the controller process to grant permissions to individual threads. When a thread asks to pass a given event label, it is placed in a corresponding queue. The controller continually looks for non-empty queues whose event labels correspond to enabled transitions from the current DFA state. When a match is found, the corresponding transition is performed and the chosen thread is resumed. Of course, the controller must be implemented to satisfy some fairness requirements. All regular trace languages can be expressed in the logic.

Applying temporal logics is a very abstract approach that can be harsh on the average programmer. However, using syntax macros, which are described in Chapter 8, it is possible to capture common concurrency primitives, such as semaphores, mutex regions, the readers/writers protocol, monitors, and so on, and provide high-level language constructs hiding the actual formulas. The advantage is that `<bigwig>` can be extended with any such constructs, even some that are highly customized to particular applications, while maintaining a simple core language for concurrency control.

The following example illustrates a simple service that implements a critical region using the event labels `enter` and `leave`:

```
service {
  shared int i;
  session Critical() {
    constraint {
      label leave,enter;
      all t1,t3: (t1<t3 && enter(t1) && enter(t3)) =>
                 is t2: t1<t2 && t2<t3 && leave(t2);
    }
    wait enter;
    i = i+1;
    wait leave;
  }
}
```

The formula states that for any two `enter` events there is a `leave` event in between, which implies that at any time at most one thread is allowed in the critical region.

Using syntax macros, programmers are allowed to build higher-level abstractions such that the following can be written instead:

```
service {
  shared int i;
  session Critical() {
    region {
      i = i+1;
    }
  }
}
```

We omit the macro definitions here. In its full generality, the `wait` statement is more like a `switch` statement that allows a thread to simultaneously attempt to pass several event labels and request a timeout after waiting a specified time.

A different example implements an asynchronous event handler. Without the macros, this could be programmed as:

```
service {
  shared int i;
  constraint {
    label handle,cause;
    all t1: handle(t1) => is t2: t2<t1 && cause(t2) &&
                                (all t3: t2<t3 && t3<t1 => !handle(t3));
  }
  session Handler() {
    while (true) {
      wait handle;
      i++;
    }
  }
  session Application() {
    wait cause;
  }
}
```

This non-trivial formula allows the handler to proceed, without blocking the application, whenever the associated event has been caused at least once since the last invocation of the handler. Fortunately, the macros again permit high-level abstractions to be introduced with more palatable syntax:

```
service {
  shared int i;
  event Increment {
    i++;
  }
  session Application() {
    cause Increment;
  }
}
```

We have dubbed the concurrency language SyCoLogic as an abbreviation of synthesizing controller logic.

7.3 Conclusion

The runtime model with a centralized controller process ensuring satisfaction of safety constraints is described in [16] and in more detail in [13]. Using a centralized process hardly qualifies as an efficient approach. However, as pointed out in [71], it is possible to analyse the constraints and distribute the safety controller. Also, in a context of relatively fast machines and comparatively slow networks, the network is likely to be the bottleneck.

The use of monadic second-order logic for controller synthesis was introduced in [71] where additionally the notions of *triggers* and *counters* are introduced to gain expressive power beyond regular sets of traces, and conditions for distributing the controller for better performance are defined.

The session model provides an opportunity to get a global view of the concurrent behavior of a service. Our current approach does not exploit this knowledge of the control flow. However, we plan to investigate how it can be used in specialized program analyses that check whether liveness and other concurrency requirements are complied with.

Chapter 8

Metamorphic Syntax Macros

8.1 Introduction

As previously mentioned, `<bigwig>` contains a notion of macros. Although not specific to Web services, this abstraction mechanism is an essential part of `<bigwig>` that serves to keep the sub-languages minimal and to tie them together.

A compiler with syntax macros accepts collections of grammatical rules that extend the syntax in which a subsequent program may be written. They have long been advocated as a means for extending programming languages [95, 22, 56]. Recent interest in domain-specific and customizable languages poses the challenge of using macros to realize new language concepts and constructs or even to grow entire new languages [79, 10, 59].

Existing macro languages are either unsafe or not expressive enough to live up to this challenge, since the syntax allowed for macro invocations is too restrictive. Also, many macro languages resort to compile-time meta-programming, making them difficult to use safely.

In this chapter we propose a new macro language that is at once sufficiently expressive and based entirely on simple declarative concepts like grammars and substitutions.

Our contributions are:

- a macro language design with guaranteed *type safety* and *termination* of the macro expansion process;
- a concept of *metamorphism* to allow a user defined grammar for invocation syntax;
- a mechanism for operating simultaneously on *multiple* parse trees;
- a full and *efficient implementation* for a syntactically rich host language; and
- a *survey* of related work, identifying and classifying relevant properties;

This work is carried out in the context of the `<bigwig>` project [74], but could find uses in many other host languages for which a top-down parser can be constructed. For a given application of our approach, knowledge of the host grammer is required. However, no special properties of such a grammar are used. In fact, it is possible to build a generator that for a given host grammar automatically will provide a parser that supports our notion of syntax macros.

8.2 Related Work

We have closely investigated the following eight macro languages and their individual semantic characteristics: the C preprocessor, `CPP` [49, 78]; the Unix macro preprocessor, `M4`; TEX's built-in macro mechanism; the macro mechanism of `Dylan` [76]; the `C++ templates` [80]; `Scheme`'s hygienic macros [48, 53]; the macro mechanism of the Jakarta Tool Suite, `JTS` [10]; and the Meta Syntactic Macro System, `MS`^{2} [95].

The JSE system [6] is a version of `Dylan` macros adapted to Java and is not treated independently here. This survey has led us to identify and group 31 properties that characterize a macro language and which we think are relevant for comparing such work. The details of our survey are presented in [19].

Our macro language shares some features of a previous work on extensible syntax [24], although that is not a macro language. Rather, it is a framework for defining new syntax that is represented as parse tree data structures in a *target* language, in which type checking and code generation is then performed. In contrast, our new syntax is directly translated into parse trees in a *host* language. Also, the host language syntax is always available on equal footing with the new syntax. However, the expressiveness of the extensible syntax that is permitted in [24] is very close to the argument syntax that we allow, although there are many technical differences, including definition selection, parsing ambiguities, expansion strategy, and error trailing. Also, we allow a more general translation scheme.

The paramount characteristic of a macro language is whether it operates at the *lexical* or *syntactical* level. Lexical macro languages allow tokens to be substituted by arbitrary sequences of characters or tokens. These definitions may be parameterized so that the substitution sequence contains placeholders for the actual parameters that are themselves just arbitrary character sequences. `CPP`, `M4`, and TEX are well-known lexical macro languages. Conceptually, lexical macro processing precedes parsing and is thus ignorant of the syntax of the underlying host language. In fact, `CPP` and `M4` are language independent preprocessors for which there is no concept of host language. As a direct consequence of syntactic independence, all lexical macro languages share many dangers that can only be avoided by clever hacks and workarounds, which are by now folklore.

A representative example is the following `square` macro:

```
#define square(X) X*X
```

which works as expected in most cases. However, if invoked with the argument `z+1` the result will be the character sequence `z+1*z+1` which is interpreted as `z+(1*z)+1`. A solution to this particular problem is explicitly to add parentheses around the arguments to control subsequent parsing:

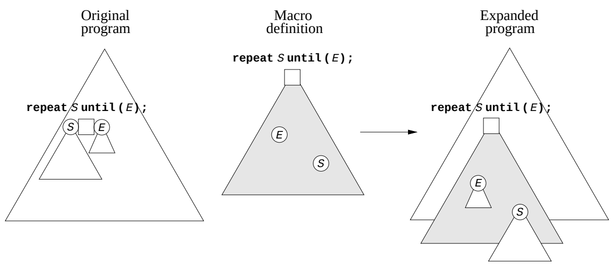


Figure 8.1: Syntax macros—operators on parse trees. The white parts are written by the service programmer and the gray parts by the macro programmer.

```
#define square(X) (X)*(X)
```

However, programmers are required to consider how individual macro invocations are being expanded and parsed. Syntactic macros amend this by operate on parse trees instead of token sequences [96]. Types are added to the macro arguments and bodies in the form of nonterminals of the host language grammar. Macro definitions can now be syntax checked at definition time, guaranteeing that parse errors no longer occur as a consequence of macro expansion. Using syntax macros, the syntax of the programming language simply appears to be extended with new productions.

In contrast, syntactical languages operate on parse trees, as depicted in Figure 8.1, which of course requires knowledge of the host language and its grammar. In our case, we have made all 55 nonterminals of a standardized version of the `<bigwig>` grammar available for extension.

8.3 Our Solution

Our macros are syntactic and based entirely on simple declarative concepts such as grammars and substitution, making them easy and safe to use by ordinary Web service programmers. Other macro languages, such as MS^2, Scheme macros, and Maya [7], instead apply full Turing complete programming languages for manipulating parse trees at compile-time, making them more difficult to use.

As an example, we will extend the core language of `<bigwig>` with a `repeat-until` control structure that is easily defined in terms of a `while` loop. Incidently, this is the macro shown in Figure 8.1.

```
macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    bool first = true;
    while (first || !<E>) {
      <S>
      first = false;
    }
  }
}
```

The first line is the header of the macro definition. It specifies the nonterminal type of the macro abstraction and the invocation syntax including the typed arguments. As expected, the type of the `repeat-until` macro is `<stm>` representing statements. This causes the body of the macro to be parsed as a statement and announces that invocations are only allowed in places where an ordinary statement would be. We allow the programmer to design the invocation syntax of the macro. This is used to guide parsing and adds to the transparency of the macro abstractions. This particular macro is designed to parse two appropriately delimited arguments, a statement `S` and an expression `E`. The body of the macro implements the abstraction using a boolean variable and a `while` loop. When the macro is invoked, the identifiers occurring in the body are α-converted to avoid name clashes with the invocation context.

8.4 Metamorphisms

Macro definitions specify two important aspects: the *syntax definitions* characterizing the syntactic structure of invocations and the *syntax transformations* specifying how "new syntax" is *morphed* into host language syntax.

In the following we will show how to move beyond a macro taking a fixed number of arguments each described by a host grammar nonterminal in a declarative way and without compromising syntactic safety. We will initially focus on the syntax definition aspects.

To present our solution and illustrate how other languages approach this, we will use an `enum` abstraction as known from C as a running example.

A first step towards greater syntactic definition flexibility is to permit the definition of macros with the same name but different invocation syntax and arguments. A notion of *specificity* selects the definition that most closely matches an invocation. Relying on a notion of specificity has the advantage of being independent of the order in which the macros are defined. This permits us to define the `enum` abstraction such that it can take one, two, or three identifier arguments:

```
macro <decls> enum { <id X> } ; ::= {
  const int <X> = 0;
}

macro <decls> enum { <id X> , <id Y> } ; ::= {
  const int <X> = 0;
  const int <Y> = 1;
}

macro <decls> enum { <id X> , <id Y> , <id Z> } ; ::= {
  const int <X> = 0;
  const int <Y> = 1;
  const int <Z> = 2;
}
```

Evidently, it is not possible to define macros with arbitrary arity and the specifications exhibit a high degree of redundancy. In terms of syntax definition, the three `enum` definitions correspond to adding three *unrelated* right-hand side productions for the nonterminal *decls*:

$$\begin{array}{lcl} \textit{decls} & : & \texttt{enum \{ } \textit{id} \texttt{ \} ;} \\ & | & \texttt{enum \{ } \textit{id} \texttt{ , } \textit{id} \texttt{ \} ;} \\ & | & \texttt{enum \{ } \textit{id} \texttt{ , } \textit{id} \texttt{ , } \textit{id} \texttt{ \} ;} \end{array}$$

`Scheme` amends this by introducing a special ellipsis construction, "`...`" to specify lists of nonterminal s-expressions. MS^2 moves one step further by permitting also tuples and optional arguments, corresponding to allowing the use of regular expressions over the terminals and nonterminals of the host grammar on the right-hand sides of productions. The ubiquitous EBNF syntax is available for designating options "`?`", lists "`*`" or "`+`", and tuples "`{...}`" (for grouping). In addition, MS^2 provides a convenient variation of the Kleene star for specifying token-separated lists of nonterminals. Here, we use $N^{\oplus}$ as notation for one-or-more *comma separated* repetitions of the nonterminal N. An `enum` macro defined via this latter construction corresponds to extending the grammar as follows:

$$\textit{decls} \quad : \quad \texttt{enum \{ } \textit{id}^{\oplus} \texttt{ \} ;}$$

The `Dylan` language has taken the full step by allowing the programmer to describe the macro invocation syntactic structure via a user defined *grammar*, permitting the introdution of new user defined nonterminals. This context-free language approach is clearly more general than the regular language approach, since it can handle balanced tree structures. The `enum` invocation syntax could be described by the following grammar fragment that introduces a user defined nonterminal called $\underline{\textit{enums}}$ (underlined for readability):

$$\begin{array}{lcl} \textit{decls} & : & \texttt{enum \{ } \textit{id} \; \underline{\textit{enums}} \texttt{ \} ;} \\ \underline{\textit{enums}} & : & \texttt{,} \; \textit{id} \; \underline{\textit{enums}} \\ & | & \varepsilon \end{array}$$

In `Dylan`, the result of parsing a user defined nonterminal also yields a result that can be substituted into the macro body. However, this result is an *unparsed* chunk of tokens with all the associated lexical macro language pitfalls.

We want to combine this great definition flexibility with type safety. Thus, we need some way of specifying and checking the *type* of the result of parsing a user defined nonterminal. Clearly, such nonterminals cannot exist on an equal footing with those of the host language; a syntax macro must ultimately produce host syntax and thus cannot return user defined ASTs. To this end, we associate to every user defined nonterminal a host nonterminal result type from which the resulting parse tree must be derived. Thus, the syntax defined by the user defined nonterminals is always morphed directly into host syntax. The specification of this morphing is inductively given for each production of the grammar. In contrast, MS^2 relies on programming and computation for specifying and transforming their regular expressions of nonterminals into parse trees.

To distinguish clearly from the host grammar, we call the user defined nonterminal productions typed with host nonterminals for *metamorphisms*. A metamorphism is a rule specifying how the macro syntax is *morphed* into host language syntax. A parameter may now also be of the form <$\underline{M}$: N a>, meaning that it is named a, has an invocation syntax that is described by the metamorph nonterminal $\underline{M}$, and that its result has type N. The metamorph syntax and the inductive translation into the host language is described by the `metamorph` rules. To the left of the "`-->`" token is the result type and name of the metamorph nonterminal, and to the right is a parameter list defining the invocation syntax and a body defining the translation into the host language. The metamorph rules may define an arbitrary grammar. In its full generality, a metamorph rule may take parse trees as arguments and produce multiple results each defined by a separate body.

We are now ready to define the general `enum` macro in our macro language. The three production rules above translates into the following three definitions:

```
macro <decls> enum { <id I> <enums: decls Ds> } ; ::= {
  int e = 0;
  const int <I> = e++;
  <Ds>
}

metamorph <decls> enums --> , <id I> <enums: decls Ds> ::= {
  const int <I> = e++;
  <Ds>
}

metamorph <decls> enums --> ::= {}
```

The first rule defines a macro `enum` with the metamorph argument <$\underline{enums}$: *decls* `Ds`> describing a piece of invocation syntax that is generated by the nonterminal *enums* in the metamorph grammar. However, *enums* parse trees are never materialized, since they are instantly morphed into parse trees of the nonterminal *decls* in the host grammar.

The body of our `enum` macro commences with the declaration of a variable `e` used for enumerating all the declared variables at runtime. This declaration is followed by the morphing of the (first) identifier `<I>` into a constant integer declaration with initialization expression `e++`. Then comes `<Ds>` which is the *decls* result of metamorphing the remaining identifiers to constant integer declarations.

The next two productions in the `enum` grammar translates into two metamorph definitions. The first will take a comma and an identifier followed by a metamorph argument and morph the identifier into a constant integer declaration as above and return this along with whatever is matched by another metamorph invocation. The second metamorph definition offers a termination condition by parsing nothing and returning the empty declarations.

For simplicity, the constant integer declarations in the bodies of the first two rules are identical. This redundance can be alleviated either by placing this constant declaration in the body of another macro or by introducing another metamorphism returning the declaration at the place of the identifiers.

```
macro <formula> allow <id L> when <formula F> ::= {
  all now: <L>(now) => restrict <F> by now;
}

macro <formula> forbid <id L> when <formula F> ::= {
  allow <L> when !<F>
}

macro <formula> mutex ( <id A> , <id B> ) ::= {
  forbid <A> when (is t: <A>(t) && (all s: t<s => !<B>(s)))
}

macro <toplevel> region <id R> ; ::= {
  constraint {
    label <R>~A, <R>~B;
    mutex(<R>~A, <R>~B);
  }
}

macro <stm> exclusive ( <id R> ) <stm S> ::= {
  { wait <R>~A;
    <S>
    wait <R>~B;
  }
}

macro <toplevels> resource <id R> ; ::= {
  region <R>;
  constraint { ... }
}

macro <stm> reader ( <id R> ) <stm S> ::= {
  { wait <R>~enterR;
    <S>
    wait <R>~exitR;
  }
}

macro <stm> writer ( <id R> ) <stm S> ::= {
  { wait <R>~P;
    exclusive (<R>) <S>
  }
}

macro <toplevels> protected <type T> <id I> ; ::= {
  <T> <I>; resource <I>;
}
```

Figure 8.2: Concurrency control abstractions.

Metamorph nonterminals are checked at definition time to intercept left-recursion so that our top-down specificity parsing terminates. Also, it is verified that possible invocations exist and that they all have a uniquely most specific match, so that the definition selection remains unambiguous.

8.5 Growing Language Concepts

Our macro language allows the host language to grow, not simply with handy abbreviations but with new concepts and constructs.

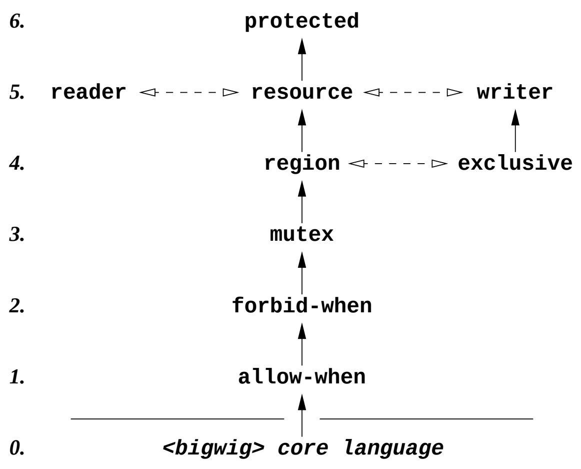


Figure 8.3: A stack of macro abstractions.

Figure 8.2 shows a whole stack of increasingly high-level concepts that are introduced on top of each other, profiting from the possibility to define macros for all nonterminals of the host language. The `allow`, `forbid`, and `mutex` macros abbreviate common constructs in temporal logic presented in Chapter 7 and produce results of type *formula*. The macro `region` of type *toplevel* is different; it introduces a new concept of *regions* that are declared on equal footing with other native concepts. The `exclusive` macro of type *stm* defines a new control structure that secures exclusive access to a previously declared region. The `resource` macro of type *toplevel_list* declares an instance of another novel concept that together with the macros `reader` and `writer` realizes the *reader/writer protocol* for specified resources. Finally, the `protected` macro seemingly provides a *modifier* that allows any declared variable to be subject to that protocol. The macros all build on top of each other and produce no less than six levels of abstraction as depicted in Figure 8.3.

An example of a program using the high-level abstractions is:

```
service {
  protected shared int counter;

  html Doc = <html>
    You are visitor number <b><[number]></b>
  </html>;

  session Access() {
    html D;
    reader (counter) D = Doc <[number=counter];
    writer (counter) counter++;
    show D;
  }
}
```

This program is a Web service that shows a page with the ubiquitous page counter which is declared using the `protected` macro. When a client issues a request to run the session `Access`, the value of the counter is read inside a `reader` region and a document showing this value is assembled. Subsequently, the counter is incremented in a `writer` region. Finally, the document is transmitted to the client.

A similar development could have implemented other primitives, such as semaphores, monitors, and fifo pipes. This demonstrates how the host language becomes highly tailorable with very simple means. The `<bigwig>` language employs an extensive collection of predefined macros to enrich the core language. They are bundled up into *packages* extending the various sub-languages of `<bigwig>` in different ways, helping to keep the `<bigwig>` language minimal. For instance, the form field validation language is extended with an *optional* and a *one-or-more* regular expression construct, and database language macros transform SQL-like queries into an iterative construction called `factor`.

8.6 Integration

Macros are also used to tie together different sub-languages, making them collaborate to provide tailor-made extensions of the language. For instance, the sub-languages dealing with sessions, dynamic documents, and concurrency control can be combined into a `publish` macro. This macro is useful when a service wishes to publish a page that is mostly static, yet once in a while needs to be recomputed, when the underlying data changes. The following macros efficiently implements such an abstraction:

```
macro <toplevels> publish <id D> { <exp E> } ::= {
  shared html <D>~cache;
  shared bool <D>~cached;
  session <D>() {
    exclusive if (!<D>~cached) {
      <D>~cache = <E>;
      <D>~cached = true;
    }
    show <D>~cache;
  }
}

macro <stm> touch <id d> ; ::= {
  <d>~cached = false;
}
```

The `publish` macro recomputes the document if the cache has expired, and then shows the document, while the `touch` macro causes the cache to expire. The `~` operator is used to create new identifiers by concatenation of others. Using this extended syntax, a service maintaining for example a high-score list can look like:

```
require "publish.wigmac"
service {
  shared int record;
  shared string holder;
  publish HiScore {
    computeWinnerDoc(record, holder)
  }
  session Play() {
    int score = play();
    if (score>=record) {
      show EnterName receive[holder=name];
      record = score;
      touch HiScore;
    } else {
      show <html>Sorry, no record.</html>;
    }
  }
}
```

Here, the high-score document is only regenerated when a player beats the record. This code is clearly easier to understand and maintain than the corresponding expanded code.

8.7 Very Domain Specific Languages: vDSL

At the University of Aarhus, undergraduate Computer Science students must complete a Bachelor's degree in one of several fields. The requirements that must be satisfied are surprisingly complicated. To guide students towards this goal, they must maintain a so-called "Bachelor's contract" that plans their remaining studies and discovers potential problems. This process is supported by a Web service that for each student iteratively accepts past and future course activities, checks them against all requirements, and diagnoses violations until a legal contract is composed. This service was first written as a straight `<bigwig>` application, but quickly became annoying to maintain due to constant changes in the curriculum. Thus it was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in a more natural language style. This makes it possible for non-programmers to maintain and update the service. An small example input is:

```
require "bachelor.wigmac"
studies
  course Math101
    title "Mathematics 101"
    2 points fall term
  ...
  course Phys202
    title "Physics 202"
    2 points spring term
  course Lab304
    title "Lab Work 304"
    1 point fall term
  exclusions
    Math101 <> MathA
    Math102 <> MathB
  prerequisites
    Math101,Math102 < Math201,Math202,Math203,Math204
    CS101,CS102 < CS201,CS203
    Math101,CS101 < CS202
    Math101 < Stat101
    CS202,CS203 < CS301,CS302,CS303,CS304
    Phys101,Phys102 < Phys201,Phys202,Phys203,Phys301
    Phys203 < Phys302,Phys303,Lab301,Lab302,Lab303
    Lab101,Lab102 < Lab201,Lab202
    Lab201,Lab202 < Lab301,Lab302,Lab303,Lab304
  field "CS-Mathematics"
    field courses
      Math101,Math102,Math201,Math202,Stat101,CS101,CS102,CS201,CS202,CS203,
      CS204,CS301,CS302,CS303, CS304,Project
    other courses
      MathA,MathB,Math203,Math204,Phys101,Phys102,Phys201,Phys202
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      at least one of Math201,Math202
      at least 2 courses among Stat101,Math202,Math203
      has 4 points among Project,CS303,CS304
      in total between 36 and 40 points
```

None of the syntax displayed is plain `<bigwig>`, except the macro package `require` instruction. The entire program is the argument to a single macro `studies` that expands into the complete code for a corresponding Web service. The file `bachelor.wigmac` is only 400 lines and yet defines a complete implementation of the new language. Thus, the `<bigwig>` macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with almost any syntax desired.

Chapter 9

Conclusion

9.1 Flexibility, Safety, and Efficiency

In this section we will conclude by investigating the thesis: that domain specific languages provide *flexible*, *safe*, and *efficient* solutions for interactive Web services.

We will do this by investigating the claim for each of the sub-languages designed for the aspects mentioned in the Introduction chapter. Also, we will show how each of these domain specific languages contribute to the overall design of the `<bigwig>` language for developing interactive Web services.

9.2 Sessions: `Runwig`

The session model and runtime system provide the *service* and *session* abstractions which are explicitly reflected in the overall structuring of `<bigwig>` services. It also offers the *show-receive* abstraction for state-preserving client interaction as well as the *flash* language feature for addressing client impatience.

Flexibility Since the runtime system automatically preserves the state across client interactions; programmers are able to *show* documents at any time, even from within deeply mutually recursively nested function calls. Without such a show abstraction, programmers would have to manually encode and save the call stack upon termination, only to decode and restore it again upon continuation.

The *flash* feature, based on the runtime system's reply indirection, enables asynchronous feedback to clients about the state of the execution. Also, the reply indirection always gives clients the freedom to bookmark a running session and continue it later. Both these features would be hard to achieve manually.

Finally, form field values submitted are automatically decoded upon reception and handed to the `<bigwig>` service. This means that programmers are never exposed to CGI encoded data.

Safety By automatically preserving the state across show statements on the server, the local state is never exposed to clients and the error-prone task of having to encode, save, decode, and restore the local state is avoided altogether.

All the problems concerning the backtracking and displaying of old interactions are avoided by the session model and the runtime system. In addition, a random interaction key protects pages from old interactions from being submitted.

Efficiency The simplicity of the single interaction paradigm permits many optimization tricks that are not possible in the session-based paradigm. However, for complex services, our implementation of the runtime system alleviates many server resources by not having to preserve local state and start new processes for every interaction.

9.3 Dynamic Documents: `DynDoc`

The `DynDoc` sub-language provides HTML/XML documents as a data type on equal footing with the other host language types. The *plug* and *show* operations are available for document construction and client interaction.

Flexibility The notion of first-class and higher-order templates with gaps enables documents to be constructed in any order; outside-in, inside-out, or any combination.

The explicit separation of programming and HTML enables programmers and designers to operate more independently. Furthermore, since documents are written in standard HTML/XML syntax, designers may write the templates using any HTML/XML authoring tools such as Microsoft FrontPage.

Safety The domain-specific static analyses provide strong compile-time safety guarantees such as interaction correspondence and validation that are not available in other languages.

Efficiency The DynDocDag datastructure provides an efficient runtime representation of documents. Furthermore, it enables the static parts to be cached on clients which saves bandwidth and server resources by performing the document unfolding on the client.

9.4 Form Field Validation: `PowerForms`

The `PowerForms` sub-language provides the concept of *regular expressions* for form field validation. It also introduces simple conditional branching based on regular expressions for specifying field interdependencies.

Flexibility Being based on a declarative formalism such as regular expressions, validation is easy to read, write, and modify. Inherently non-operational, it focuses on *what* to do as opposed to *how* to do it and is thus available to a wider audience; even non-programmers. Also clients benefit from our approach by automatically getting the validation incrementally and with visual feedback.

Safety All the problems pertaining to diverging and incomplete JavaScript implementations in different browsers are completely eliminated. The compiler generates code that uses only a simple subset of JavaScript that is known to work correctly in all common browser implementations.

The `PowerForms` compiler also automatically generates server-side validation identical to that performed on the client-side. Additionally, the server-side revalidation is augmented to double check that forms are not tampered with; for instance, that the `maxlength` attribute of text fields is not bypassed and that options submitted from selection widgets were indeed available.

Efficiency The JavaScript implementation works by interpreting minimized deterministic finite automata and is thus highly efficient. This could possibly be sped up even further by inlining the automaton in the JavaScript control flow. Finally, performing the validation incrementally on the client saves time, bandwidth, and cycles.

9.5 Concurrency Control: `SyCoLogic`

The `SyCoLogic` concurrency sub-language is integrated in the `<bigwig>` language through the *wait* statement, label declarations, and monadic second-order logic.

Flexibility Our approach presents programmers with a uniform way of dealing with all concurrency control aspects while separating the service code and the safety logic. This means that the safety logic may be added independently to constrain the overall behaviour of the service and is pervious to changes made in the service code.

Like regular expressions, monadic second-order logic is a declarative formalism that focuses on *what* to do as opposed to *how* to do it, rendering the safety requirements easier to maintain.

Finally, the logic is succinct in that a formula may be non-elementarily smaller than its operational counterpart, the minimized deterministic finite automaton.

Safety The safety controller employes queues and a token ring strategy to ensure that no session threads waiting at enabled checkpoints are blocked indefinitely.

Efficiency Using a centralized process hardly qualifies as an efficient approach. However, as pointed out in [71], it is possible to analyse the constraints and distribute the safety controller. Also, in a context of relatively fast machines and comparatively slow networks, the network is likely to be the bottleneck.

9.6 Metamorphic Syntax Macros

The syntax macros in `<bigwig>` can not really be classified as a DSL, but serve to extend the language and glue the many sub-languages together. Also, they may be used to create whole new domain specific languages.

The syntax macros provide a uniform abstraction mechanism for language extension that works alike for all syntactic categories of the host language. It is based entirely on declarative concepts such as pattern matching and substitution. The metamorphisms provide a flexible invocation syntax based on grammars along with an inductive type safe transformation into host language syntax.

The macros are geared towards extensibility, providing a notion of specificity that resolves grammar ambiguities locally and in a way that disregards the order in which macros are defined.

The macros are *syntactically safe*, checked at definition-time to guarantee termination and that no syntax errors occur as a result of macro expansion. Automatic alpha conversion avoids identifier name clashes. Also, metamorphic grammars are checked to ensure that our specificity resolution always has a unique final winner and that possible invocations exist for all the macros defined.

9.7 Domain Specific Languages for Interactive Web Services

Domain specific languages applied to the domain of interactive Web services achieve flexible, safe, and efficient solutions for sessions, documents, forms, and concurrency. These sub-languages may be integrated into a host language and conveniently tied together through metamorphic syntax macros. The result is a domain specific language, `<bigwig>`, that provides support for virtually all aspects of the development of interactive Web services.

Part II

Publications

Chapter 10

The <bigwig> Project

with Anders Møller and Michael I. Schwartzbach

Abstract

We present the results of the <bigwig> project, which aims to design and implement a high-level domain-specific language for programming interactive Web services.

A fundamental aspect of the development of the World Wide Web during the last decade is the gradual change from static to dynamic generation of Web pages. Generating Web pages dynamically in dialogue with the client has the advantage of providing up-to-date and tailor-made information. The development of systems for constructing such dynamic Web services has emerged as a whole new research area.

The <bigwig> language is designed by analyzing its application domain and identifying fundamental aspects of Web services inspired by problems and solutions in existing Web service development languages. The core of the design consists of a session-centered service model together with a flexible template-based mechanism for dynamic Web page construction. Using specialized program analyses, certain Web specific properties are verified at compile-time, for instance that only valid HTML 4.01 is ever shown to the clients. In addition, the design provides high-level solutions to form field validation, caching of dynamic pages, and temporal-logic based concurrency control, and it proposes syntax macros for making highly domain-specific languages.

The language is implemented via widely available Web technologies, such as Apache on the server-side and JavaScript and Java Applets on the client-side. We conclude with experience and evaluation of the project.

10.1 Introduction

The <bigwig> project was founded in 1998 at the BRICS Research Center at the University of Aarhus to design and implement a high-level domain-specific language for programming interactive Web services. In the following we will argue that existing Web service programming languages in various ways provide only low-level solutions to problems specific to the domain of Web services. Our overall ambitions of the project are to identify the key areas of the Web service domain, analyze the problems with the existing approaches, and provide high-level solutions that will support development of complex services.

10.1.1 Motivation

Specifically, we will look at the following Web service technologies: the HTTP/CGI Web protocol [39], Sun's Java Servlets [81] and their JavaServer Pages (JSP) [82], Microsoft's Active Server Pages (ASP) [41], the related Open Source language PHP [5], and the research language MAWL [4, 3, 55].

CGI was the first platform for development of Web services. It is based on the simple idea of letting a script generate the reply to incoming HTTP requests dynamically on the server, rather than returning a static HTML page from a file. Typically, the script is written in the general-purpose scripting language Perl, but any language supported by the server can be used. Being based on general-purpose programming languages, there is no special support for Web specific tasks, such as generation of HTML pages, and knowledge of the low-level details of the HTTP protocol are required. Also, HTTP/CGI is a stateless protocol that by itself provides no help for tracking and guiding users through series of individual interactions. This can to some degree be alleviated by libraries. In any case, there are no compile-time guarantees of correct run-time behavior when it comes to Web specific properties, for instance ensuring that invalid HTML is never sent to the clients.

Servlets are a popular higher-level Java-specific approach. Servlets, which are special Java programs, offers the common Java advantages of network support, strong security guarantees, and concurrency control. However, some significant problems still exist. Services programmed with servlets consist of collections of request handlers for individual interactions. Sessions consisting of several interactions with the same client must be carefully encoded with cookies, URL rewriting, or hidden input fields, which is tedious and error prone, even with library support, and it becomes hard to maintain an overview of large services with complex interaction flows. A second, although smaller, problem is that state shared between multiple client sessions, even for simple services, must be explicitly stored in a name–value map called the "servlet context" instead of using Java's standard variable declaration scoping mechanism. Thirdly, the dynamic construction of Web pages is not improved compared to CGI. Web pages are built by printing string fragments to an output stream. There is no guarantee that the result always becomes valid HTML. This situation is slightly improved by using HTML constructor libraries, but they preclude the possibility of dividing the work of the programmers and the HTML designers. Furthermore, since client sessions are split into individual interactions that are only combined implicitly, for instance by storing session IDs in cookies, it is not possible to statically analyze that a given page sent to a client always contains exactly the input fields that the next servlet in the session expects.

Both JSP, ASP, PHP, and the countless homegrown variants were designed from a different starting point. Instead of aiming for complex services where all parts of the pages are dynamically generated, they fit into the niche where pages have mostly static contents and only little fragments are dynamically generated. A service written in one of these languages typically consists of a collection of “server pages” which are HTML pages with program code embedded in special tags. When such a page is requested by the client, the code is evaluated and replaced by the resulting string. This gives better control over the HTML construction, but it only gives an advantage for simple services where most of every page is static.

The MAWL language was designed especially for the domain of interactive Web services. One innovation of MAWL is to make client sessions explicit in the program logic. Another is the idea of building HTML pages from templates. A MAWL service contains a number of sessions, shared data, and HTML templates. Sessions serve as entry points of client-initiated session threads. Rather than producing a single HTML page and then terminating as CGI scripts or Servlets, each session thread may involve multiple client interactions while maintaining data that is local to that thread. An HTML template in MAWL is an HTML document containing named gaps where either text strings or special lists may be inserted. Each client interaction is performed by inserting appropriate data into the gaps in an HTML template, and then sending it to the client who fills in form fields and submits the reply back to the server.

The notions of sessions and document templates are inherent in the language and being compilation-based it allows important properties to be verified statically without actually running the service. Since HTML documents are always constructed from the templates, HTML validity can be verified statically. Also, since it is clear from the service code where execution resumes when a client submits form input, it can be statically checked that the input fields match what the program expects. One practical limitation of the MAWL approach is that the HTML template mechanism is quite restrictive as one cannot insert markup into the template gaps.

We describe more details of these existing languages in the following sections. By studying services written in any of these language, some other common problems show up. First of all, often surprisingly large portions of the service code tend to deal with form input validation. Client-server interaction takes place mainly through input forms, and usually some fields must be filled with a certain kind of data, perhaps depending on what has been entered in other fields. If invalid data is submitted, an appropriate error message must be returned so that the client can try again. All this can be handled either on the client-side—typically with JavaScript [35], in the server code, or with a combination. In any case, it is tedious to encode.

Secondly, one drawback of dynamically generated Web pages compared to static ones is that traditional caching techniques do not work well. Browser caches and proxy servers can cause major improvements in saving network bandwidth, load time, and clock cycles, but when moving towards interactive Web services, these benefits disappear.

Thirdly, most Web services act as interfaces to underlying databases that for instance contain information about customers, products, and orders. Accessing databases from general-purpose programming languages where database queries are not integrated requires the queries to be built as text strings that are sent to a database engine. This means that there is no static type checking of the queries. As known from modern programming languages, type systems allow many programming bugs to be caught at compile-time rather than at run-time, and thereby improve reliability and reduce development cost.

Fourthly, since running Web services contain many concurrently executing threads and they access shared information, for instance in databases on the server, there is a fundamental need for concurrency control. Threads may require exclusive access to critical regions, be blocked until certain events occur, or be required to satisfy more high-level behavioral constraints. All this while the service should run smoothly without deadlocks and other abrupt obstacles. Existing solutions typically provide no or only little support for this, for instance via low-level semaphores as Perl or synchronized methods in Servlets. This can make it difficult to guarantee correct concurrent execution of entire services.

Finally, since Web services usually operate on the Internet rather than on secure local networks, it is important to protect sensitive information both from hostile attacks and from programming leaks. A big step forward is the Secure Sockets Layer (SSL) protocol [36] combined with HTTP Authentication [11]. These techniques can ensure communication authenticity and confidentiality, but using them properly requires insight of technical protocol and implementation details. Furthermore, they do not protect against programming bugs that unintentionally leak secret information. The "taint mode" in Perl offers some solution to this. However, it is run-time based so no compile-time guarantees are given. Also, it only checks for certain predefined properties, and more specialized properties cannot be added.

10.1.2 The `<bigwig>` Language

Motivated by the languages and problems described above we have identified the following areas as key aspects of Web service development:

- *sessions*: the underlying paradigm of interactive Web services;
- *dynamic documents*: HTML pages must be constructed in a flexible, efficient, and safe fashion;
- *concurrency control*: Web services consist of collections of processes running concurrently and sharing resources;
- *form field validation*: validating user input requires too much attention of Web programmers so a higher-level solution is desirable;
- *database integration*: the core of a Web service is often a database with a number of sessions providing Web access; and
- *security*: to ensure authenticity and confidentiality, both regarding malicious clients and programming bugs.

To attack the problems we have from scratch designed a new language called `<bigwig>`, as a descendant of the MAWL language. This language is a high-level, domain-specific language [89], meaning that it employs special syntax and constructs that are tailored to fit its particular application domain and allow specialized program analyses, in contrast to library based solutions. Its core is a C or Java-like skeleton, which is surrounded by domain-specific sub-languages covering the above key aspects. A notion of *syntax macros* tie the sub-languages together and provide additional layers of abstraction. This macro language, which operates on the parse tree level, rather that the token sequence level as conventional macro languages, has proved successful in providing extensions of the core language. This has helped each of the sub-languages remain minimal, since desired syntactic sugar is given by the macros. Syntax macros can be taken to the extreme where they with little effort can define a completely new syntax for *very*-domain-specific languages tailored to highly specialized application domains.

It is important that `<bigwig>` is based on compilation rather than on interpretation of a scripting language. Unlike many other approaches, we can then apply type systems and static analysis to catch many classes of errors before the service is actually installed.

The `<bigwig>` compiler uses common Web technologies as target languages. This includes HTML [68], HTTP [11], JavaScript [35], and Java Applets [2]. Our current implementation additionally relies on the Apache Web server. It is important to apply only standard technologies on the client-side in order not to place restrictions on the clients. In particular, we do not use browser plug-ins, and we only use the subset of JavaScript that works on all common browsers. As new technologies become standard, the compiler will merely obtain corresponding opportunities for generating better code. To the degree it is possible, we attempt to hide the low-level technical details of the underlying technologies.

We have made no effort to contribute to the graphical design of Web services. Rather, we provide a clean separation between the physical layout of the HTML pages and the logical structure of the service semantics. Thus, we expect that standard HTML authoring tools are used, conceivably by others than the Web programmer. Also, we do not focus on efficiency, but on providing higher levels of abstraction for the developers. For now, we regard it as less important to generate solutions that seamlessly scale to thousands of interactions per second, although scalability of course is an issue for the design.

The main contributions of the `<bigwig>` project are the following results:

- The notion of client sessions can and should be made explicit in Web service programming languages;
- dynamic construction of Web pages can be made at the same time flexible and fast while still permitting powerful compile-time analyses;
- form field validation can be made easier with a domain-specific language based on regular expressions and boolean logic;
- temporal logic is a useful formalisms for expressing concurrency constraints and synthesizing safety controllers; and
- syntax macros can be used to create very-domain-specific high-level languages for extremely narrow application domains.

We focus on these key contributions in the remainder of this paper, but also describe less central contributions, such as a technique for performing client-side caching of dynamically generated pages, a built-in relational database, and simple security mechanisms. The individual results have been published in previous more specialized papers [71, 16, 72, 15, 17, 14, 19]. Together, these results show that there is a need for high-level programming languages that are tailor-made to the domain of Web service development.

10.1.3 Overview

We begin in Section 10.2 by classifying the existing Web service languages as either script-, page-, or session-centered, arguing for the latter as the best choice for complex services. In Section 10.3, we show how the HTML template mechanism from MAWL can be extended to become more flexible using a notion of higher-order templates. Using novel type systems and static analyses the safety benefits of MAWL templates remain, in spite of the increased expressibility. Also, we show how our solution can be used to cache considerable parts of the dynamically generated pages in the browser. In Section 10.4, we address the problem of validating form input more easily. Section 10.5 describes a technique for generating concurrency controllers from temporal logic specifications. Section 10.6 gives an introduction to the syntax macro mechanism that ties together the sub-languages of `<bigwig>`. In Section 10.7, we mention various less central of the `<bigwig>` language. Finally, in Section 10.8 we describe our implementation and a number of applications, and evaluate various practical aspects of `<bigwig>`.

10.2 Session-Centered Web Services

Web programming covers a wide spectrum of activities, from composing static HTML documents to implementing autonomous agents that roam the Web. We focus in our work on *interactive Web services*, which are Web servers on which clients can initiate sessions that involve several exchanges of information mediated by HTML forms. This definition includes large classes of well-known services, such as news services, search engines, software repositories, and bulletin boards, but also covers services with more complex and specialized behavior.

There are a variety of techniques for implementing interactive Web services. These can be divided into three main paradigms: the *script-centered*, the *page-centered*, and the *session-centered*. Each is supported by various tools and suggests a particular set of concepts inherent to Web services.

10.2.1 The Script-Centered Approach

The script-centered approach builds directly on top of the plain, stateless HTTP/CGI protocol. A Web service is defined by a collection of loosely related scripts. A script is executed upon request from a client, receiving form data as input and producing HTML as output before terminating. Individual requests are tied together by explicitly inserting appropriate links to other scripts in the reply pages.

A prototypical scripting language is Perl, but almost any programming language has been suggested for this role. CGI scripting is often supported by a large collection of library functions for decoding form data, validating input, accessing databases, and realizing semaphores. Even though such libraries are targeted at the domain of Web services, the language itself is not. A major problem is that the overall behavior is distributed over numerous individual scripts and depends on the implicit manner in which they pass control to each other. This design complicates maintenance and precludes any sort of automated global analysis, leaving all errors to be detected in the running service [34, 3].

HTML documents are created on the fly by the scripts, typically using `print`-like statements. This again means that no static guarantees can be issued about their correctness. Furthermore, the control and presentation of a service are mixed together in the script code, and it is difficult to factor out the work of programmers and HTML designers [29].

The Java Servlets language also fits into this category. The overall structure of a service written with servlets is the same as for Perl. Every possible interaction is essentially defined by a separate script, and one must use cookies, hidden input fields, or similar techniques to connect sequences of interactions with the clients. Servlets provide a session tracking API that hides many of the details of cookies, hidden input fields, and URL rewriting. Many servlet servers use cookies if the browser supports them, but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled. This API is exemplified by the following code inspired by two Servlet tutorials[1]:

```
public class SessionServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
        throws ServletException, IOException {
    ServletContext context =  getServletContext();
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><HEAD><TITLE>Servlet Demo</TITLE></HEAD><BODY>");
    if (session.isNew()) {
      out.println("<FORM ACTION=SessionServlet>" +
                  "Enter your name: <INPUT NAME=handle>" +
                  "<P><INPUT TYPE=SUBMIT></FORM>");
      session.putValue("state", "1");
    } else {
      String state = (String) session.getValue("state");
      if (state.equals("1")) {
        String name = (String) request.getParameter("handle");
        int users =
          ((Integer) context.getAttribute("users")).intValue() + 1;
        context.setAttribute("users", new Integer(users));
        session.putValue("name", name);
        out.println("<FORM ACTION=SessionServlet>" +
```

[1]`http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/` and `http://java.sun.com/docs/books/tutorial/servlets/`

```
                "Hello " + name + ", you are user number " + users +
                "<P><INPUT TYPE=SUBMIT></FORM>");
          session.putValue("state", "2");
        } else /* state.equals("2") */ {
          String name = (String) session.getValue("name");
          out.println("Goodbye " + name);
          session.invalidate();
        }
      }
      out.println("</BODY></HTML>");
    }
  }
```

Clients running this service are guided through a series of interactions: first, the service prompts for the client's name, then the name and the total number of invocations is shown, and finally a "goodbye" page is shown. The `ServletContext` object contains information shared to all sessions, while the `HttpSession` object is local to each session. The code is essentially a `switch` statement that branches according to the current interaction. An alternative approach is to make a servlet for each kind of interaction. In spite of the API, one still needs to explicitly maintain both the state and the identity of the session.

The model of sessions that is supported by Servlets and other script-centered approaches tends to fit better with "shopping basket applications" where the client browses freely among dynamically generated pages, than with complex services that need to impose more strict control of the interactions.

10.2.2 The Page-Centered Approach

The page-centered approach is covered by language such as ASP, PHP, and JSP, where the dynamic code is embedded in the HTML pages. In a sense, this is the inverse of the script-centered languages where HTML fragments are embedded in the program code. When a client requests a page, a specialized Web server interprets the embedded code, which typically produces additional HTML snippets while accessing a shared database. In the case of JSP, implementations work by compiling each JSP page into a servlet using a simple transformation.

This approach is often beautifully motivated by simple examples, where pages are mainly static and only sporadically contain computed contents. For example, a page that displays the time of day or the number of accesses clearly fits this mold. The following JSP page dynamically inserts the current time together with a title and a user name based on the CGI input parameters:

```
<HTML><HEAD><TITLE>JSP Demo</TITLE></HEAD><BODY>
Hello <%
  String name = request.getParameter("who");
  if (name==null) name = "stranger";
  out.print(name);
%>!
<P>
This page was last updated: <%= new Date() %>
</BODY></HTML>
```

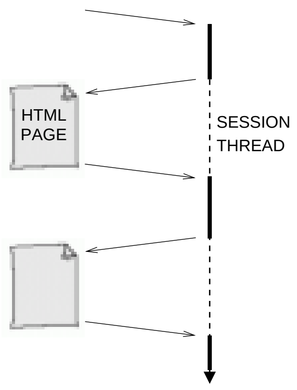


Figure 10.1: Client-server sessions in Web services. On the left is the client's browser, on the right is a session thread running on the server. The thread is initiated by a client request and controls the sequence of interactions.

The special `<%...%>` tags contain Java code that is evaluated at the time of the request. As long as the code parts only generate strings without markup it is easy to statically guarantee that all shown pages are valid HTML and other relevant properties. But as the services become more complex, the page-centered approach tends to converge towards the script-centered one. Instead of a mainly static HTML page with some code inserted, the typical picture is a single large code tag that dynamically computes the entire contents. Thus, the two approaches are closely related, and the page-centered technologies are only superior to the degree in which their scripting languages are better designed.

The ASP and PHP languages are very reminiscent of JSP. ASP is closely tied to Microsoft's Internet Information Server, although other implementations exist. Instead of being based on Java it defines a language-independent connection between HTML pages and scripting languages, typically either Visual Basic Script or Microsoft's version of JavaScript. PHP is a popular Open Source variant whose scripting language is a mixture of C, Java, and Perl.

These languages generally provide only low-level support for tracking client sessions and maintaining session state. Cookies, hidden input fields, and some library support is the common solution. Also for other Web service aspects, such as databases and security, there is often a wide range of libraries available but no direct language support.

10.2.3 The Session-Centered Approach

The pure session-centered approach was pioneered by the MAWL project. A service is here viewed as a collection of distinct *sessions* that access some shared data. A client may initiate a session *thread*, which is conceptually a process running on the server. Interaction with the client is viewed as remote procedure calls from the the server, as known from classical construction of distributed systems but with the roles reversed.

The flow of an entire session is programmed as a single sequential program, which is closer to ordinary programming practice and offers the compiler a chance to obtain a global view of the service. Figure 10.1 illustrates the flow of control in this approach. Important issues such as concurrency control become simpler to understand in this context and standard programming solutions are more likely to be applicable.

The following MAWL program is equivalent to the previous Servlet example:

```
static int users = 0;

session GreetingSession {
  auto form {} -> {handle} hello;
  auto string name = hello.put().handle;

  auto form {string who, int count} -> {} greeting;
  users++;
  greeting.put({name, users});

  auto form {string who} -> {} goodbye;
  goodbye.put({name});
}
```

The HTML templates *hello*, *greeting*, and *goodbye* are placed in separate files. Here is `hello.mhtml`:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Enter your name: <INPUT NAME=handle>
</BODY></HTML>
```

and `greeting.mhtml`:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Hello <MVAR NAME=who>, you are user number <MVAR NAME=count>
</BODY></HTML>
```

The template for *goodbye* is similar. A form tag and a continue button are implicitly inserted. Variables declared `static` contain persistent data, while those declared `auto` contain per-session data. The `form` variables are declared with two record types. The former defines the set of gaps that occur in the template, and the latter defines the input fields. In the templates, gaps are written with `MVAR` tags. Template variables all have a `put` method. When this is executed, the arguments are inserted in the gaps, the resulting page is sent to the client who fills in the fields and submits the reply, which is turned into a record value in the program. Note how the notion of sessions is explicit in the program, that private and shared state is simply a matter of variable declaration modifiers, and that the templates are cleanly separated from the service logic. Obviously, the session flow is more clear, both to the programmer and to the compiler, than with the non-session based approaches. One concrete benefit is that it is easy to statically check both validity and correct use of input fields.

The main force of the session-centered approach is for services where the control flow is complex. Many simple Web services are in actuality more loosely structured. If all sessions are tiny and simply does the work of a server module from the page-centered approach, then the overhead associated with sessions may seem to large. Script-centered services can be seen as a subset of the session-centered where every session contains only one client interaction. Clearly, the restriction in the script-centered and the page-centered languages allow significant performance improvements. For instance, J2EE Servlet/JSP servers employ pools of short-lived threads that store only little local state. For more involved services, however, the session-centered approach makes programming easier since session management comes for free.

10.2.4 Structure of `<bigwig>` Services

The overall structure of `<bigwig>` programs is directly inspired by MAWL. A `<bigwig>` program contains a complete specification of a Web *service*. A service contains a collection of named *sessions*, each of which essentially is an ordinary sequential program. A client has the initiative to invoke a thread of a given session, which is a process on the server that executes the corresponding sequential code and exclusively communicates with the originating client. Communication is performed by *showing* the client an HTML page, which implicitly is made into a form with an appropriate URL return address. While the client views the given document, the session thread is suspended on the server. Eventually the client submits the form, which causes the session thread to be resumed and any form data entered by the client to be *received* into program variables. A simple `<bigwig>` service that communicates with a client as in the Servlet and MAWL examples is the following:

```
service {
  html hello = <html>Enter your name: <input name=handle></html>;

  html greeting =
    <html>Hello <[who]>, you are user number <[count]></html>;

  html goodbye = <html>Goodbye <[who]></html>;

  shared int users = 0;

  session Hello() {
    string name;
    show hello receive[name=handle];
    users++;
    show greeting<[who=name,count=users];
    show goodbye<[who=name];
  }
}
```

The program structure is obviously as in MAWL, except that the session code and the templates are wrapped into a `service` block. For instance, the `show-receive` statements produce the client interactions similarly to the `put` methods in MAWL. However, `<bigwig>` provides a number of new features. Most importantly, HTML templates are now *first-class values.* That is, `html` is a built-in data type, and its values can be passed around and stored in variables as for any other data type. Also, the HTML templates are *higher-order.* This means that instead of only allowing text strings to be inserted into the template gaps, we also allow insertion of other templates. This is done with the the special *plug* operator, x`<[`y=z`]` which inserts a string or template z into the y gaps of the x template. Clearly, this constitutes a more flexible document construction mechanism, but it also calls for new ideas for statically verifying for instance HTML validity. This is the topic of Section 10.3. Other new features include the techniques for improving form field validation and concurrency control, together with the syntax macro mechanism, all of which are described in the following sections.

10.2.5 A Session-Based Runtime Model

The session-based model can be implemented on top of the CGI protocol. One naive approach is to create session threads as CGI scripts where all local state is stored on disk. At every session interaction, the thread must be started again and restore its local state, including the call stack, in order to continue execution. A better approach is to implement each session thread as a process that runs for the whole duration of the session. For every interaction, a tiny transient CGI script called a *connector process* is executed, acting as a pipe between the Web server and the session process. This approach resembles FastCGI [66] and is described in detail in [16]. Our newest implementation is instead based on a specialized Apache server module[2]. Naturally, this is much faster than the CGI solutions since it does not create a new process for every single interaction, but only for the session processes.

Two common sources of problems with standard implementations of sessions are the history buffers and the bookmarking features found in most browsers. With the history buffers and the "back" button, the users can step back to a previous interaction, and either intentionally or unintentionally resubmit an old input form. Sometimes this can be a useful feature, but more often this causes confusion and annoyance to the users who may for instance order something twice. It is a general problem that the information shown to the user in this way can be obsolete since it was tailor-made only for the exact time of the initial request. Since the information was generated from a shared database that may have changed entirely, it does generally not make sense to "step back in time" using the history buffer. This is no different from ordinary programs. Even if the programmer has been aware of this and has added serial number checks, the history buffer will be full of URLs to obsolete requests. If the service really needs a "back" feature, it can be programmed explicitly into the flow of the sessions. It also becomes hazardous to try to use bookmarks to temporarily suspend a session. Invoking the bookmark will then typically cause a CGI script to be executed a second time instead of just displaying its results again.

`<bigwig>` provides a simple but unique solution to these problems: Each session thread is associated a URL which points to a file on the server containing the latest HTML page shown to the client. Instead of sending the contents directly to the client at every `show` statement, we redirect the browser to this URL, as illustrated in Figure 10.2. Since the URL serves as the identification of the session thread, this solves the problems mentioned above: The history

[2]See `http://httpd.apache.org/`.

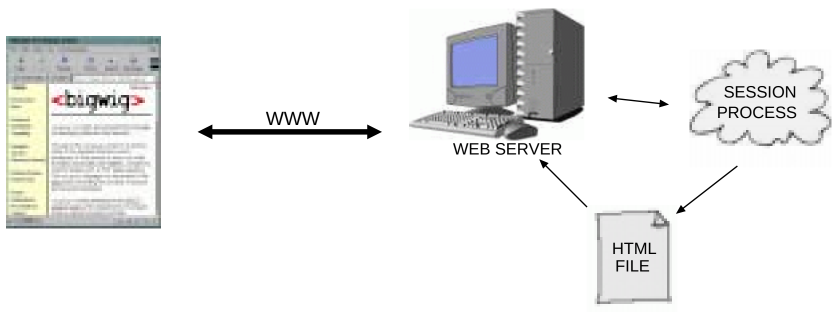


Figure 10.2: Session-based runtime model with reply indirection. Each session thread is implemented as a separate process that writes its HTML reply to a designated file.

list of the browser now only contains a single entry for the duration of the session, the sessions can now be bookmarked for later use, and in addition the session identity URL can be passed around manually—to another browser for instance—without problems. When using URLs instead of cookies to represent the session identity it also becomes possible for a single user to simultaneously run multiple sessions in different windows but with the same browser.

With this simple solution we can furthermore automatically provide the client with feedback while the server is processing a request. This is done by after a few seconds writing a temporary response to the HTML file, which informs the client about the status of the request. This temporary file reloads itself frequently, allowing for updated status reports. When the final response is ready, it simply overwrites the temporary reply file, causing the reloading to stop and the response to be shown. This simple technique may prevent the client from becoming impatient and abandoning the session.

The `<bigwig>` runtime system additionally contains a garbage collector process that monitors the service and shuts down session processes which have been abandoned by the clients. By default, this occurs if the client has not responded within 24 hours. The sessions are allowed to execute some clean-up actions before terminating.

10.3 Dynamic Construction of HTML Pages

In MAWL, all HTML templates are placed in separate files and viewed as a kind of procedures, with the arguments being strings that are plugged into gaps in the template and the results being the values of the form fields that the template contains. This allows a complete separation of the service code and the HTML code. Two benefits are that static guarantees are possible, and that the work of programmers and HTML designers can be separated, as previously mentioned. A disadvantage is that the template mechanism becomes too rigid compared to the flexibility of the `print`-like statements available in the script-centered languages. However those languages permit essentially no static guarantees or work separation. Furthermore, with the script-centered solutions the HTML must often be constructed in a linear fashion from top to bottom, instead of being composed from components in a more logical manner. The `<bigwig>`

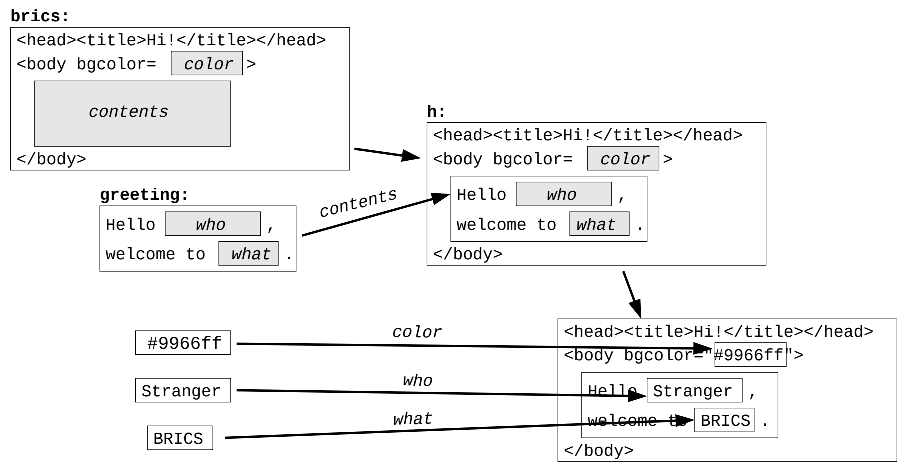


Figure 10.3: Building a document by plugging into template gaps. The construction starts with the five constants on the left and ends with the complete document on the right.

solution provides the best from the two worlds. Higher-order HTML templates as first-class values are in practice as flexible as `print` statements, and still the MAWL benefits are preserved.

We define *DynDoc* as the sub-language of `<bigwig>` that deals with document construction, that is, the control structures, HTML template constants, variables and assignments, plug operations, and `show-receive` statements. Template constants are delimited by `<html>`...`</html>`. Gaps are written with special `<[`...`]>` tags. Special *attribute gaps* can be used in place of attribute values, as shown in the example below. Of course, only strings should be plugged into such gaps, not templates with markup. The plug operation x`<[`y=z`]` creates a new template by inserting a copy of z in the y gaps of a copy of x. When used in a `show-receive` statement, a template is converted to a complete document by implicitly plugging empty strings into all remaining gaps. Also, it is automatically wrapped into a `form` element whose action is to continue the session, unless the session terminates immediately after. And finally, it is inserted into an outermost template like:

```
<html><head><title>service</title></head><body>...</body></html>
```

unless already inside a `body` element. The following example illustrates that documents can be built gradually using higher-order templates:

```
service {
  html brics = <html>
    <head><title>Hi!</title></head>
    <body bgcolor=[color]><[contents]></body>
  </html>;
  html greeting = <html>Hello <[who]>, welcome to <[what]>.</html>;
  session Welcome() {
    html h = brics<[contents=greeting];
    show h<[color="#9966ff",who="Stranger",what="BRICS"];
  }
}
```

The construction process is shown in Figure 10.3. Note that gaps may be plugged in any order, not necessarily "bottom up". MAWL does provide little functionality beyond plugging text strings into gaps. The special `MITER` tag allows list structures to be built iteratively, but that still precludes general tree-like structures. The following `<bigwig>` example uses a recursive function to construct an HTML document representing a binary tree:

```
service {
  html list = <html><ul><li><[gap]><li><[gap]></ul></html>;
  html tree(int i) {
    if (i==0) return <html>foo</html>;
    return list<[gap=tree(i-1)];
  }
  session ShowTree() {
    show tree(10);
  }
}
```

Something similar could not be done with MAWL's first-order templates. In a script-centered or a page-centered language it is of course possible, but not with such a simple program structure reflecting the logical composition of the document, because it must be generated linearly by printing to the output stream. An alternative is to use an HTML tree constructor library, however, that forces documents to be built bottom up which is often inconvenient.

The use of higher-order templates generally leads to programs with a large number of relatively small template constants. For that reason it is convenient to be able to inline the constants in the program code, as in these examples, rather than always placing them in separate files. However, we do offer explicit support for factoring out the work of graphical designers using a `#include` construct like in C. Alternatively, each HTML constant appearing in a `<bigwig>` program may have associated a URL pointing to an alternate, presumably more elaborate version:

```
service {
  session Hello {
    show <html>Hello World</html> @ "fancy/hello.html";
  }
}
```

The compiler retrieves the indicated file and uses its contents in place of the constant, provided it exists and contains well-formed HTML. In this manner, the programmer can use plain versions of the templates while a graphical designer simultaneously produces fancy versions. The compiler checks that the two versions have the same gaps and fields. In order to accommodate the use of HTML authoring tools, we permit gaps to be specified in an alternative syntax using special tags.

The DynDoc sub-language was introduced in [72] where it is also shown how this template model can be implemented efficiently with a compact runtime representation. The plug operation takes only constant time, and showing a document takes time linear in the size of the output. Also, the size of the runtime representation of a document may be only a fraction of its printed size. For example, a binary tree of height n shown earlier has a representation of size $O(n)$ rather than $O(2^n)$.

10.3.1 Analysis of Template Construction and Form Input

We wish to devise a type checker that allows as liberal a use of dynamic documents as possible, while guaranteeing that no errors occur. More precisely, we would like to verify the following properties at compile-time:

- at every plug operation, x`<[`y`=`z`]`, there always exists a y gap in x;
- the gap types are compatible with the values being plugged in, in particular, HTML with markup tags is never inserted into attribute gaps;
- for every `show-receive` statement, the fields in the `receive` part always exist in the document being shown;
- the field types are compatible with the `receive` parts, for instance, a select menu allowing multiple items to be selected yields a vector value; and
- only valid HTML 4.01 [68] is ever sent to the clients.

The first four properties are addressed in [72] as summarized in the following. The last property is covered in the next section.

It is infeasible to explicitly declare the exact types of higher-order templates for two reasons. Firstly, all gaps and all fields and their individual capabilities would have to be described, which may become rather voluminous. Secondly, this would also imply that an HTML variable has the same type at every program point, which is too restrictive to allow templates to be composed in an intuitive manner. Consequently, we rely instead on a flow analysis to infer the types of template variables and expressions at every program point. In our experience, this results in a liberal and useful mechanism.

We employ a monovariant interprocedural flow analysis, which guarantees that the form fields in a shown document correspond to those that are received, and that gaps are always present when they are being plugged. This analysis fits into standard data-flow frameworks [65], however it applies a highly specialized lattice structure representing the template types. For every template variable and expression that occurs in the given program, we associate a lattice element that abstractly captures the relevant template properties. The lattice consists of two components: a *gap map* and a *field map*. The gap map records for every occurring gap name whether or not the gap occurs at that point, and in case it does occur, whether it is an HTML gap or an attribute gap. Similarly, the field map records for every occurring input field name information about the input fields, which can be of type text, radio, select, or checkbox, representing the different interaction methods.

Given a `<bigwig>` program we construct a flow graph. This is quite easy since there are no higher-order functions or virtual methods. All language

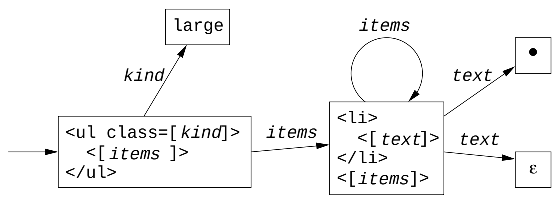


Figure 10.4: A summary graph representing a set of HTML fragments.

constructs that are not included in DynDoc are abstracted away. It is now possible to define transfer functions which abstractly describe the effect of the program statements. This produces a constraint system which we solve using a classical fixed point iteration technique. From this solution, we inspect that the first three properties mentioned above are satisfied, and if not, generate error messages indicating the cause.

With this approach, the programmer is only restricted by the requirement that at every program point, the template type of an expression must be fixed. In practice, this does not limit the expressibility, rather, it tends to enforce a more comprehensible structure of the programs. Also, the compiler silently resolves conflicts at flow join points by implicitly plugging superfluous gaps with empty contents.

10.3.2 HTML Validity Analysis

The fifth property, HTML validity, is addressed with a similar but more complicated approach as described in [17].

The main idea is the following: We define a finite structure called a *summary graph* that approximates the set of templates that a given HTML expression may evaluate to. This structure contains the plug operations and the constant templates and strings that are involved.

As an example, consider the summary graph in Figure 10.4. The nodes correspond to program constants, and the edges correspond to plug operations. For instance, the `li` template may here be plugged into the *`items`* gaps in the `ul` template. The • node represents arbitrary text strings and ϵ is the empty string. The root of the graph corresponds to the outermost template. By "unfolding" this graph according to the plug edges, this summary graph defines a possibly infinite set of HTML fragments without gaps, in this case the set of all `ul` lists of class `large` with one or more character data items. This structure turns out to provide an ideal abstraction level for verifying HTML validity.

Again, we apply a data-flow analysis to approximate the flow of template values in the program. This time we use a lattice consisting of summary graphs. It is possible to model plug operations with good precision using transfer functions, however two preliminary analyses are required. One for tracking string constants, and one, called a *gap track analysis*, for tracking the origins of gaps. The latter tells us for each template variable and gap name, which constant templates containing such a gap can flow into that variable at any given program point. Clearly, these analyses are highly specialized for the domain of dynamic document construction and for `<bigwig>`'s higher-order template mechanism, but they all fit into the standard data-flow analysis frameworks. For more details we refer to [17].

Once we have the summary graphs for all the `show` statements, we need to verify that the sets of document fragments they define all are valid HTML according to W3C's official definition. To simplify the process we reformulate the notion of Document Type Definition (DTD) as a simpler and more convenient formalism that we call *abstract DTD*. An abstract DTD consists of a number of *element declarations* whereof one is designated as the root. An element declaration defines the requirements for a particular type of elements. Each declaration consists of an element name, a set of names of attributes and subelements that may occur, and a boolean expression constraining the element type instances with respect to their attribute values and contents. The official DTD for HTML is easily rewritten into our abstract DTD notation. In fact, the abstract DTD version captures more validity requirements than those expressible by standard DTDs and merely appear as comments in the HTML DTD. As a technicality we actually work with XHTML 1.0 which is an XML reformulation of HTML 4.01. There are no conceptual differences, except that the XML version provides a cleaner tree view of documents for the analysis.

Given a summary graph and an abstract DTD description of HTML, validity can be checked by a recursive traversal of the summary graph starting at the roots. We memoize intermediate results to ensure termination since the summary graphs may contain loops. If no violations are encountered, the summary graph is valid. Since all validity properties are local to single elements and their contents, we are able to produce precise error messages in case of violations. Analysis soundness is ensured by the following property: if all summary graphs corresponding to `show` expressions are verified to be valid with respect to the abstract DTD, then all concrete documents are guaranteed to be valid HTML.

The program analyses described here all have high worst-case complexities because of the complex lattices. Nevertheless, our implementations and experiments show that they work well in practice, even for large intricate programs. These experiments are mentioned in Section 10.8.

10.3.3 Caching of Dynamically Generated HTML

Traditional Web caching based on HTTP works by associating an expiration time to all documents sent by the servers to the clients. This has helped in decreasing both network and server load and response times. By default, no expiration is set, and by using "now", caching is effectively disabled. This technique was designed primarily for documents whose contents rarely or never changes, not for documents dynamically generated by interactive Web services. The gradual change from statically to dynamically generated documents has therefore caused the impact of Web caching to degrade.

Existing proposals addressing this include Active Cache, HPP, and various server-based techniques, as explained in the survey in [14]. Server-based techniques aim for relieving the server of redundant computations, not for decreasing network load. They typically work by simplifying assumptions, for instance that many interactions can be handled without side-effects on the global service state, that interactions are often identical for many clients, or that the dynamics of the pages is limited to e.g. banner ad rotation. None of this applies to complex interactive services. Active Cache is a proxy-based solution that employs programmable cache applets. This can be very effective, but it requires both specialized proxy servers and careful programming to ensure consistency between the proxies and the main server.

HPP tries to separate the constant parts from the dynamic parts of the generated documents. We apply a similar technique. In contrast to HPP, our solution is entirely automatic while HPP requires extra programming. The idea is to exploit the clear division between the service code and the HTML templates present in `<bigwig>`. In our normal implementation of DynDoc, the internal template representation is converted to an HTML document on the server when the `show` statement is executed. Instead, we now store each template constant in a fixed file on the server, and defer the conversion to the client using a JavaScript representation of the dynamic parts. The template files can now be cached by the ordinary browser caches. More details of the technique can be found in [14]. We summarize our evaluation results in Section 10.8.

10.3.4 Code Gaps and Document Clusters

In the following, we describe two extensions to the DynDoc language. Occasionally, the page-centered approach is admittedly more appropriate than the session-centered. Consider the following example, which gives the current time of day:

```
service {
  session Time() {
    html h = <html>Right now, the time is <[t]></html>;
    show h<[t=now()];
  }
}
```

An equivalent but less clumsy version can be written using *code gaps*, which implicitly represent expressions whose values are computed and plugged into gaps when the document is being shown:

```
service {
  session Time() {
    html h = <html>Right now, the time is <[(now())]></html>;
    show h;
  }
}
```

Documents with code gaps remain first-class values, since the code can only access the global scope. Note that code gaps in `<bigwig>` are more powerful than the usual page-centered approach, since the code exists in the full context of sessions, shared variables, and concurrency control. In fact, with the idea of *published* documents described in Section 10.6, the page-centered approach is now included as a special case of `<bigwig>`.

Some services may want to offer the client more than a single document to browse, for example, the response could be a tiny customized Web site. In `<bigwig>` we have experimented with support for showing such *document clusters*. The difficulty is to provide a simple notation for specifying an arbitrary graph of documents connected by links. We introduce for an HTML variable `x` the *document reference* notation `&x` which can be used as the right-hand side of a plug operation. It will eventually expand into a URL, but not until the document is finally shown. Until then, the flow analysis just records the connection between the gap and the variable. When a document is shown, the transitive closure of document references is computed, and the resulting cluster of documents is produced with references replaced by corresponding URLs. The following example shows a cluster of two documents that are cyclically connected. Notice that the cluster can be browsed freely without cluttering the control-flow:

```
service {
  session Cluster() {
    html greeting = <html>
      Hi! Click <a href=[where]>here</a> for a kind word.
    </html>;
    html kind = <html>How nice to see you! <a href=[there]>Back</a></html>;
    kind = kind<[there = &Greeting];
    show greeting<[where=&kind];
  }
}
```

The compiler checks that all cluster documents with submit buttons contain the same form fields. It is also necessary to perform an escape analysis to ensure that document variables are not exported out of their scope.

10.4 Form Field Validation

A considerable effort in Web programming is expended on form field validation, that is, checking whether the data supplied by the client in form fields are valid, and when it is not, producing error messages and requesting the fields to be filled in again. Apart from details about regular expression matching, the main problem is to program a solution that is robust, efficient, and user friendly.

One approach is *server-side* validation, where the form fields are validated on the server when the page has been submitted. None of the languages mentioned in Section 10.1 provides any help for this, except the regular expression matching in Perl. Therefore, the main logic of the service often becomes cluttered with validation code. In a sense, every program part that sends a page to a client must be wrapped into a while-loop that repeats until the input is valid. Other disadvantages include wasting bandwidth and causing delays to the users. One proposal addressing some of these problems without requiring browser extensions or Java applets is

The alternative is *client-side* validation, which usually requires the programmer to use JavaScript in the pages being generated. This permits more sophisticated user interactions and reduces the communication overhead. However, client-side validation should not be used alone. The reason is that the client is perfectly capable of bypassing the JavaScript code, so an additional server side validation must always be performed. Thus, the same code must essentially be written both in JavaScript and in the server scripting language. In practice, writing JavaScript input validators that at the same time capture all validity requirements and also are user friendly can be very difficult since most browsers unfortunately differ in their JavaScript support. Whole Web sites are dedicated to explaining how the various subsets of JavaScript work in different browsers[3].

In `<bigwig>` we have introduced a domain-specific sub-language, called PowerForms, for form field validation [15]. It handles complex interdependencies between form fields, and the compiler generates the required code for both client and server. By compiling into JavaScript, only the PowerForms implementors need to know the details of how browsers support JavaScript, rather than all Web service programmers. Also, the programmer needs not anymore write essentially the same code in a server-side version and a client-side version.

PowerForms is a declarative language. Informally, this means that the programmer specifies what the input requirements are, not how to check them. In its simplest form, PowerForms allows regular-expression *formats* to be associated to form fields:

```
service {
  format Digit = range('0','9');
  format Number = plus(Digit);
  format Alpha = union(range('a','z'),range('A','Z'));
  format Word = concat(Alpha,star(union(Digit,Alpha)));
  format Name = concat(Word,star(concat(" ",Word)));
  format Email = concat(Word,"@",Word,star(concat(".",Word)));
  session Validate() {
    html form = <html>
      Please enter your email address:
      <input name=email type=text size=20>
      <format name=Email field=email>
    </html>;
    string s;
    show Form receive[s=email];
  }
}
```

This example shows how to constrain input in the `email` field to a certain regular expression. The `<bigwig>` compiler generates the JavaScript code that checks the user input on the client-side and provides help and error messages, and also the code performing the server-side double-check. Using "traffic-light" icons next to the input fields, the user is provided with continuous feedback about the string entered so far. "Green" means valid, "yellow" means invalid but prefix of something valid, and "red" means not prefix of something valid. Other alternatives can be chosen, such as checkmark symbols, arrows, etc. We

[3]See e.g. `http://www.webdevelopersjournal.com/articles/javascript_limitations.html` or `http://www.xs4all.nl/~ppk/js/version5.html`.

also allow the usual Perl-style syntax for regular expressions in the subset of our notation that excludes the intersection and complement operators.

Formats can be associated to all kinds of form fields, not just those of type `text`. For `select` fields, the format is used to filter the available options. For `radio` and `checkbox` fields, only the permitted buttons can be depressed.

As noted in [32], many forms contain fields whose values may be constrained by those entered in other fields. A typical example is a field that is not applicable if some other field has a certain value. Such interdependencies are almost always handled on the server, even if the rest of the validation is performed on the client. The reason is presumably that interdependencies require even more delicate JavaScript programming. The `<bigwig>` solution is to allow such field interdependencies to be specified using an extension of the regular expressions: the `format` tags are extended to describe boolean decision trees, whose conditions probe the values of other form fields and whose leaves are simple formats. The interdependence is resolved by a fixed-point process that is computed on the client by JavaScript code automatically generated by the `<bigwig>` compiler. A simple example is the following, where the client chooses a letter group and the `select` menu is then dynamically restricted to those letters:

```
service {
  format Vowel = charset("aeiouy");
  format Consonant = charset("bcdfghjklmnpqrstvwxz");
  html form = <html>
    Favorite letter group:
    <input type=radio name=group value=vowel checked>vowels
    <input type=radio name=group value=consonant>consonants
    <br>
    Favorite letter:
    <select name=letter>
      <option value="a">a
      <option value="b">b
      <option value="c">c
      ...
      <option value="z">z
    </select>
    <format field=letter>
       <if><equal field=group value=vowel>
       <then><format name=Vowel></then>
       <else><format name=Consonant></else>
       </if>
    </format>
  </html>;
  session Letter() {
    string s;
    show form receive[s=letter];
  }
}
```

ColdFusion [21] provides a mechanism reminiscent of PowerForms. However, it does not support field interdependencies or validation of non-`text` fields. PowerForms have shown to be a simple language with a clean semantics that appears to handle most realistic situations. We have implemented it both as part of the `<bigwig>` compiler and as a stand-alone tool that can be used to add input validation to general HTML pages.

10.5 Concurrency Control

As services have several session threads, there is a need for synchronization and other concurrency control to discipline the concurrent behavior of the active threads. A simple case is to control access to the shared variables using mutex regions or the readers/writers protocol. Another issue is enforcement of priorities between different session kinds, such that a management session may block other sessions from running. Another example is event handling, where a session thread may wait for certain events to be caused by other threads.

We deal with all of these scenarios in a uniform manner based on a central controller process in the runtime system, which is general enough to enforce a wide range of safety properties [71]. The support for concurrency control in the previously mentioned Web languages is limited to more traditional solutions, such as file locking, monitor regions, or synchronized methods.

A `<bigwig>` service has an associated set of *event labels*. During execution, a session thread may request permission from the controller to pass a specific event checkpoint. Until such permission is granted, the session thread is suspended. The policy of the controller must be programmed to maintain the appropriate global invariants for the entire service. Clearly, this calls for a domain-specific sub-language. We have chosen a well-known and very general formalism, temporal logic. In particular, we use a variation of monadic second-order logic [87]. A formula describes a set of strings of event labels, and the associated semantics is that the trace of all event labels being passed by all threads must belong to that set. To guide the controller, the `<bigwig>` compiler uses the MONA tool [50] to translate the given formula into a minimal deterministic finite-state automaton that is used by the controller process to grant permissions to individual threads. When a thread asks to pass a given event label, it is placed in a corresponding queue. The controller continually looks for non-empty queues whose event labels correspond to enabled transitions from the current DFA state. When a match is found, the corresponding transition is performed and the chosen thread is resumed. Of course, the controller must be implemented to satisfy some fairness requirements. All regular trace languages can be expressed in the logic.

Applying temporal logics is a very abstract approach that can be harsh on the average programmer. However, using syntax macros, which are described in Section 10.6, it is possible to capture common concurrency primitives, such as semaphores, mutex regions, the readers/writers protocol, monitors, and so on, and provide high-level language constructs hiding the actual formulas. The advantage is that `<bigwig>` can be extended with any such constructs, even some that are highly customized to particular applications, while maintaining a simple core language for concurrency control.

The following example illustrates a simple service that implements a critical region using the event labels `enter` and `leave`:

```
service {
  shared int i;
  session Critical() {
    constraint {
      label leave,enter;
      all t1,t3: (t1<t3 && enter(t1) && enter(t3)) =>
                 is t2: t1<t2 && t2<t3 && leave(t2);
    }
    wait enter;
    i = i+1;
    wait leave;
  }
}
```

The formula states that for any two `enter` events there is a `leave` event in between, which implies that at any time at most one thread is allowed in the critical region. Using syntax macros, programmers are allowed to build higher-level abstractions such that the following can be written instead:

```
service {
  shared int i;
  session Critical() {
    region {
      i = i+1;
    }
  }
}
```

We omit the macro definitions here. In its full generality, the `wait` statement is more like a `switch` statement that allows a thread to simultaneously attempt to pass several event labels and request a timeout after waiting a specified time.

A different example implements an asynchronous event handler. Without the macros, this could be programmed as:

```
service {
  shared int i;
  constraint {
    label handle,cause;
    all t1: handle(t1) => is t2: t2<t1 && cause(t2) &&
                                 (all t3: t2<t3 && t3<t1 => !handle(t3));
  }
  session Handler() {
    while (true) {
      wait handle;
      i++;
    }
  }
  session Application() {
    wait cause;
  }
}
```

This non-trivial formula allows the handler to proceed, without blocking the application, whenever the associated event has been caused at least once since the last invocation of the handler. Fortunately, the macros again permit high-level abstractions to be introduced with more palatable syntax:

```
service {
  shared int i;
  event Increment {
    i++;
  }
  session Application() {
    cause Increment;
  }
}
```

The runtime model with a centralized controller process ensuring satisfaction of safety constraints is described in [16]. The use of monadic second-order logic for controller synthesis was introduced in [71] where additionally the notions of *triggers* and *counters* are introduced to gain expressive power beyond regular sets of traces, and conditions for distributing the controller for better performance are defined.

The session model provides an opportunity to get a global view of the concurrent behavior of a service. Our current approach does not exploit this knowledge of the control flow. However, we plan to investigate how it can be used in specialized program analyses that check whether liveness and other concurrency requirements are complied with.

10.6 Syntax Macros

As previously mentioned, `<bigwig>` contains a notion of macros. Although not specific to Web services, this abstraction mechanism is an essential part of `<bigwig>` that serves to keep the sub-languages minimal and to tie them together.

A macro language can be characterized by its level of operation which is either *lexical* or *syntactic*. Lexical macro languages operate on sequences of tokens and conceptually precede parsing. Because of this independence of syntax, macros often have unintended effects, and parse errors are only discovered at invocation time. Consequently, programmers are required to consider how individual macro invocations are being expanded and parsed. Syntactic macros amend this by operate on parse trees instead of token sequences [96]. Types are added to the macro arguments and bodies in the form of nonterminals of the host language grammar. Macro definitions can now be syntax checked at definition time, guaranteeing that parse errors no longer occur as a consequence of macro expansion. Using syntax macros, the syntax of the programming language simply appears to be extended with new productions.

Our macros are syntactic and based entirely on simple declarative concepts such as grammars and substitution, making them easy and safe to use by ordinary Web service programmers. Other macro languages, such as MS^2, Scheme macros, and Maya, instead apply full Turing complete programming languages for manipulating parse trees at compile-time, making them more difficult to use.

As an initial example, we will extend the core language of `<bigwig>` with a `repeat-until` control structure that is easily defined in terms of a `while` loop.

```
macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    bool first = true;
    while (first || !<E>) {
      <S>
      first = false;
    }
  }
}
```

The first line is the header of the macro definition. It specifies the nonterminal type of the macro abstraction and the invocation syntax including the typed arguments. As expected, the type of the `repeat-until` macro is `<stm>` representing statements. This causes the body of the macro to be parsed as a statement and announces that invocations are only allowed in places where an ordinary statement would be. We allow the programmer to design the invocation syntax of the macro. This is used to guide parsing and adds to the transparency of the macro abstractions. This particular macro is designed to parse two appropriately delimited arguments, a statement `S` and an expression `E`. The body of the macro implements the abstraction using a boolean variable and a `while` loop. When the macro is invoked, the identifiers occurring in the body are α-converted to avoid name clashes with the invocation context.

With a concept of *packages*, macros can be bundled up in collections. Our experience with `<bigwig>` programming has led us to develop a "standard macro package", `std.wigmac`, that extends the sub-languages of `<bigwig>` in various ways and has helped keep the language minimal. For instance, the form field validation language is extended with an `optional` regular expression construct, and database language macros transform SQL-like queries into our own iterative `factor` construction. Also, various composite security modifiers are defined, and concurrency control macros, such as the `region` from Section 10.5, gradually build on top of each other to implement increasingly sophisticated abstractions.

Macros are also used to tie together different sub-languages, making them collaborate to provide tailor-made extensions of the language. For instance, the sub-languages dealing with sessions, dynamic documents, and concurrency control can be combined into a `publish` macro. This macro is useful when a service wishes to publish a page that is mostly static, yet once in a while needs to be recomputed, when the underlying data changes. The following macros efficiently implements such an abstraction:

```
macro <toplevels> publish <id D> { <exp E> } ::= {
  shared html <D>~cache;
  shared bool <D>~cached;
  session <D>() {
    exclusive if (!<D>~cached) {
      <D>~cache = <E>;
      <D>~cached = true;
    }
    show <D>~cache;
  }
}
```

```
macro <stm> touch <id d> ; ::= {
  <d>~cached = false;
}
```

The `publish` macro recomputes the document if the cache has expired, and then shows the document, while the `touch` macro causes the cache to expire. The ~ operator is used to create new identifiers by concatenation of others. Using this extended syntax, a service maintaining for example a high-score list can look like:

```
require "publish.wigmac"
service {
  shared int record;
  shared string holder;
  publish HiScore {
    computeWinnerDoc(record, holder)
  }
  session Play() {
    int score = play();
    if (score>=record) {
      show EnterName receive[holder=name];
      record = score;
      touch HiScore;
    } else {
      show <html>Sorry, no record.</html>;
    }
  }
}
```

Here, the high-score document is only regenerated when a player beats the record. This code is clearly easier to understand and maintain than the corresponding expanded code.

The expressive power of syntax macros is extended with a concept of *metamorphisms*, as explained in [19]. This declaratively permits tree structures to be transformed into host language syntax without compromising syntactic safety, something not possible with other macro languages. Using this mechanism in an extreme way, it is possible to define whole new languages. We call this concept a *very* domain-specific language, or VDSL.

At the University of Aarhus, undergraduate Computer Science students must complete a Bachelor's degree in one of several fields. The requirements that must be satisfied are surprisingly complicated. To guide students towards this goal, they must maintain a so-called "Bachelor's contract" that plans their remaining studies and discovers potential problems. This process is supported by a Web service that for each student iteratively accepts past and future course activities, checks them against all requirements, and diagnoses violations until a legal contract is composed. This service was first written as a straight `<bigwig>` application, but quickly became annoying to maintain due to constant changes in the curriculum. Thus it was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in a more natural language style. This makes it possible for non-programmers to maintain and update the service. An small example input is:

```
require "bachelor.wigmac"
studies
  course Math101
    title "Mathematics 101"
    2 points fall term
  ...
  course Phys202
    title "Physics 202"
    2 points spring term
  course Lab304
    title "Lab Work 304"
    1 point fall term
  exclusions
    Math101 <> MathA
    Math102 <> MathB
  prerequisites
    Math101,Math102 < Math201,Math202,Math203,Math204
    CS101,CS102 < CS201,CS203
    Math101,CS101 < CS202
    Math101 < Stat101
    CS202,CS203 < CS301,CS302,CS303,CS304
    Phys101,Phys102 < Phys201,Phys202,Phys203,Phys301
    Phys203 < Phys302,Phys303,Lab301,Lab302,Lab303
    Lab101,Lab102 < Lab201,Lab202
    Lab201,Lab202 < Lab301,Lab302,Lab303,Lab304
  field "CS-Mathematics"
    field courses
      Math101,Math102,Math201,Math202,Stat101,CS101,CS102,CS201,CS202,CS203,
      CS204,CS301,CS302,CS303, CS304,Project
    other courses
      MathA,MathB,Math203,Math204,Phys101,Phys102,Phys201,Phys202
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      at least one of Math201,Math202
      at least 2 courses among Stat101,Math202,Math203
      has 4 points among Project,CS303,CS304
      in total between 36 and 40 points
```

None of the syntax displayed is plain `<bigwig>`, except the macro package `require` instruction. The entire program is the argument to a single macro `studies` that expands into the complete code for a corresponding Web service. The file `bachelor.wigmac` is only 400 lines and yet defines a complete implementation of the new language. Thus, the `<bigwig>` macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with almost any syntax desired. Similar features do not occur in any of the Web service languages mentioned in the previous sections.

10.7 Other Web Service Aspects

There are of course other features in `<bigwig>` that are necessary to support Web service development but for which we have no major innovations. These are briefly presented in this section.

10.7.1 HTML Deconstruction

The template mechanism is used to construct HTML documents, but when "run in reverse" it also allows for deconstruction. This is realized by using the templates as patterns in which the gaps play the role of variables, as illustrated in this example:

```
service {
  html Template = <html>
    <[]><img src=[source] alt="today's Dilbert comic"><[]>
  </html>;
  session Dilbert() {
    string data = get("http://www.dilbert.com/");
    string s;
    match(data,Template)[s =source];
    exit Template<[source ="http://www.dilbert.com"+s];
  }
}
```

which grabs the daily strip from the Dilbert home page. Gaps without names serve as wildcards.

10.7.2 Seslets

For some interaction patterns, a strict session model can be inappropriate since the client and server must alternate between being active and suspended. Furthermore, information cannot be pushed on the server's initiative while the client is viewing a page. A simple example is a chat room where new messages should appear automatically, without the client having to reload the page being viewed, and where only the new message and not the entire new page is transmitted. The essence of this concept is *client-side computations* that are able to contact the server on their own accord.

The `<bigwig>` solution is a notion of *seslets*. A seslet is a kind of light-weight session which is allowed to do anything an ordinary session can do, except perform `show` operations. It is invoked by the client with some arguments and eventually returns a reply of any `<bigwig>` type. Typically, it performs database operations or waits for certain events to occur, and reports back to the client.

Since we are limited by the existing technologies on the client-side, our current implementation is restricted to using Java applets or JavaScript. To facilitate the writing of applets, the `<bigwig>` compiler generates the Java code for an abstract class extending `Applet`, which must be inherited from in order to access the available seslets. Alternatively, we have experimented with a JavaScript interface. However, this approach is limited by the lack of client-server communication support from JavaScript, so we currently apply cookies for the communication.

An important use of seslets is to allow client-side code to synchronize with other active threads on the server. For example, the chat room solution could employ a seslet that uses the concurrency control mechanisms of `<bigwig>` to wait until the next message is available, which then is returned to the applet. In this way, no client pulling or busy waiting is required.

10.7.3 Databases

Most Web services are centered around a database. In the general case, this is an existing, external database which the service must connect to. The `<bigwig>` system supports the ODBC interface for this purpose. However, queries are not built as strings but are written in a query language that is part of the `<bigwig>` syntax. This allows for compile-time checking of the syntax and types of queries, eliminating another source of errors. Since many smaller services use only simple data, we also offer an internal database that is implemented on top of the file system.

10.7.4 Security

There are many aspects of Web service security[4]. The security in `<bigwig>` can be divided into two categories, depending on whether it is generically applicable to all services or specific to the behavior of a particular service.

The former category mostly relates to the runtime environment and communication, dealing with concepts such as integrity, authenticity, and confidentiality. Integrity of a session thread's local state is achieved by keeping it exclusively on the server. Integrity of shared data is provided by the database. An interaction key is generated and embedded in every document shown to the client and serves to prevent submission of old documents. Clients and session threads are associated through a random key which is created by the server upon the first request and carried across interactions in a hidden input field. This mechanism may optionally be combined with other security measures, such as SSL, to provide the necessary level of security. Authenticity and confidentiality is addressed through general declarative security modifiers that the programmer can attach on a `service`, `session`, or `show` basis. The modifiers `ssl` and `htaccess` enforce that the SSL and HTTP Authentication protocols are used for communication. The `selective` modifier restricts access to a session to those clients whose IP numbers match a given set of prefixes. Finally, the `singular` modifier ensures that the client has the same IP address throughout the execution of a session.

We envision performing some simple static analyses relating to the behavioral security of particular services. Values are classified as *secret* or *trusted*, and, in contrast to tainting in Perl, the compiler keeps track of the propagation of these properties. Furthermore, there are restrictions on how each kind of data can be used. Form data is always assumed to be untrusted and gaps are never allowed to be plugged with secret values. Variables can be declared with the modifiers `secret` or `trusted` and may then only contain the corresponding values. The `system` function can only be called with a trusted string value. To change the classification of a value, there are two functions `trust` and `disclose`. The programmer must make the explicit choice of using these coercions. An example involving trust is the following service:

```
service {
  session Lookup() {
```

[4]http://www.w3.org/Security/faq/

```
    html Error = <html>Invalid URL!</html>;
    html EnterURL = <html>Enter a URL: <input type=text name=URL></html>;
    string u,domain;
    show EnterURL receive[u = URL];
    if (|u|<7 || u[0..7]!="http://") show Error;
    for (i=7; i<|u| && u[i]!='/'; i++);
    domain = u[7..i];
    if (system("/usr/sbin/nslookup '" + domain + "'").stderr!="") {
      show Error;
    }
  }
}
```

This code performs an *nslookup* on the URL supplied by the user to check whether its domain exists. Since the value of `domain` is derived from the form field `URL` it should not be trusted, and its use in the call of `system` will be flagged by the compiler. And, indeed, it would be unfortunate if the client enters `"http://foo';rm -rf /'"` in the form. A similar analysis is performed for `secret`. Consider the example:

```
service {
  shared secret string password;
  bool odd(int n) { return n%2==1; }
  session Reveal() {
    if (odd(|password|)) show <html>foo</html>;
  }
}
```

The compiler is sufficiently paranoid to reject this program, since the branching of the `if`-statement depends on a function applied to information derived from a secret value. These analyses are not particularly original, but are not seen in other Web service programming languages.

There is still much work to be done in this area. So far, we have not considered using cryptological techniques to ensure service integrity, the role of certificates, or more sophisticated static analyses.

10.8 Evaluation

The `<bigwig>` language should be evaluated according to two different criteria. First, the quality of our language design as seen from concrete programming experiences. This is necessarily a somewhat intangible and subjective criterion. Second, the performance of our language implementation as seen from observed benchmarks.

10.8.1 Experience with `<bigwig>`

`<bigwig>` is still mainly an experimental research tool, but we have gained experiences from numerous minor services that we have written for our own edification, a good number of services that are used for administrative purposes at the University of Aarhus, and a couple of production services on which we have collaborated. Apart from these applications, we estimate that `<bigwig>`

has been downloaded roughly 2500 times from our Web site, and we have mainly received positive feedback from the users.

One production service is the Web site of the European Association for Theoretical Computer Science (`www.eatcs.org`), handling newsletters, webboards, and several membership services. It is written in 5,345 lines of `<bigwig>`, using 133 HTML templates, and 114 `show` statements. Another is the Web site of the JAOO 2001 conference (`www.jaoo.dk`), handling all aspects of advertisement, schedules, registration, and attendant services. It is written is 7,943 lines of `<bigwig>`, using 248 HTML templates, and 39 `show` statements.

These experiences have shown that `<bigwig>` has two very strong points. First, the session concept greatly simplifies the programming of complicated control flow with multiple client interactions. Second, the HTML templates are very easy and intuitive to use and the static guarantees catch numerous errors, many of which are difficult to find by other means. It is particularly helpful that the HTML analyzers provide precise and intuitive error messages.

The JAOO application has been particularly interesting, since it involved collaboration with an external HTML designer. This experience confirmed that our templates are successful in defining an interface between programmers and designers and that gaps and fields define a useful contract between the two.

The main weak point that we have identified is the core language which is often found to be lacking minor features. We plan to address this in future work, as mentioned in Section 10.9.

The stand-alone version of the PowerForms sub-language has been surprisingly popular in its own right. It has many active users, and has been integrated into a proprietary Web deployment system.

10.8.2 Performance

When evaluating the performance of the `<bigwig>` implementation, we want to focus on the areas where we have attempted to provide improvements. We are not aiming for simple high-load services, but are focusing on services with intricate flow of control. Still, informal tests show that the throughput of our services is certainly comparable with that of straight CGI-based services or Servlet applications running on J2SE.

The automatic caching scheme based on our HTML templates is designed to exploit their intricate structure to cache static fragments on the client-side. We have obtained real benefits from this approach. The experiments reported in [14] show that the size of the transmitted data may shrink by a factor of 3 to 30, which on a dial-up connections translates into a reduction in download times by a factor of 2 to 10.

It is also relevant to evaluate the performance of the `<bigwig>` compiler, since we employ a series of theoretically quite expensive static analyses. However, in practice they perform very well, as documented in [72, 17]. The EATCS service is analyzed for HTML validity in 6.7 seconds and the JAOO service in 2.4 seconds.

10.9 Conclusion

The <bigwig> project has identified central aspects of interactive Web services and has provided solutions in a coherent framework based on programming language theory. At the same time, the <bigwig> project is a case study in applications of the domain-specific language design paradigm.

We have argued that the notion of sessions is essential to Web services and should constitute the basic structure of a Web service programming language. Together with higher-order document templates, such as in the DynDoc sub-language, the dynamic construction of Web pages becomes at the same time flexible, making it is easy to use, and safe, through compile-time guarantees regarding document validity and the use of input forms. We have shown that form field validation, compared to traditional approaches, can be made easier with a domain-specific sub-language, such as PowerForms, which automatically translates high-level specifications into a combination of more low-level server-side and client-side code. We have examined how temporal logics can be use to synthesize concurrency controllers. Finally, we have demonstrated how macro mechanisms can be made effective for extending and combining languages, in the context of the sub-languages of <bigwig>.

Version 2.0 of the <bigwig> compiler and runtime system is freely available from the project home page at www.brics.dk/bigwig/ where documentation and examples also can be found.

Regarding the future development of <bigwig> we now move towards Java. We are developing JWIG [26] as an extension of Java where we add the most successful features of <bigwig>, such as the session model, dynamic documents, form field validation, and syntax macros. Since the design of <bigwig> has focused on the Web specific areas, we hope that the many standard programming issues of Web services become easier to develop with JWIG. However, a number of new challenges arise. For instance, the program analyses described in Section 10.3 all assume that we have access to precise control-flow graphs of the programs. This is trivial for <bigwig>, but certainly not for Java. Other future plans include type-safe support for XML document transformation, WML and VoiceXML support, and broadening the view towards development and management of whole Web sites comprising many services.

10.9.1 Acknowledgments

Tom Ball provided us with extensive and very helpful information about experiences with the MAWL language. Anders Sandholm has been a key participant during his PhD studies at BRICS. Mikkel Ricky Christensen and Steffan Olesen have worked tirelessly as student programmers during the entire project. Niels Damgaard, Uffe Engberg, Mads Johan Jurik, Lone Haudrum Olesen, Christian Stenz, and Tommy Thorn have provided valuable feedback and suggestions. We also appreciate the efforts made by the participants of the WIG Projects course in Spring 1998. Finally, we are grateful for the insightful comments we have received from the anonymous reviewers.

Chapter 11

A Runtime System for Interactive Web Services

with Anders Møller, Anders Sandholm, and Michael I. Schwartzbach

Abstract

Interactive web services are increasingly replacing traditional static web pages. Producing web services seems to require a tremendous amount of laborious low-level coding due to the primitive nature of CGI programming. We present ideas for an improved runtime system for interactive web services built on top of CGI running on virtually every combination of browser and HTTP/CGI server. The runtime system has been implemented and used extensively in `<bigwig>`, a tool for producing interactive web services.

11.1 Introduction

An interactive web service consists of a global shared state (typically a database) and a number of distinct sessions that each contain some local private state and a sequential, imperative action. A web client may invoke an individual thread of one of the given session kinds. The execution of this thread may interact with the client and inspect or modify the global state.

One way of providing a runtime system for interactive web services would be to simply use plain CGI scripts [39]. However, being designed for much simpler tasks, the CGI protocol by itself is inadequate for implementing the session concept. It neither supports long sessions involving many user interactions nor any kind of concurrency control. Being the only widespread standard for running web services, this has become a serious stumbling stone in the development of complex modern web services.

We present in this paper a runtime system built on top of the CGI protocol that among other features has support for sessions and concurrency control. First, we motivate the need for a runtime system such as the one presented here. This is done by presenting its advantages over a simple CGI script based solution. Afterwards, a description of the runtime system, its different parts, and its dynamic behavior is given. We round off with a discussion of related work, a conclusion, and directions for future work.

In the appendices, we briefly describe an implementation of the suggested runtime system. Also, we give a short presentation of `<bigwig>` [74], which is a tool for producing interactive web services that makes extensive use of the self-contained runtime system package.

11.2 Motivation

The technology of plain CGI scripts lacks several of the properties one would expect from a modern programming environment. In the following we discuss various shortcomings of traditional CGI programming and motivate our solution to these problems, namely the design of an improved runtime system built on top of the standard CGI protocol.

11.2.1 The session concept

First, we will describe and motivate the concept of an interactive web service.

The HTTP protocol was originally designed for browsing *static* documents connected with hyperlinks. CGI together with forms allows *dynamic* creation of documents, that is, the contents of a document are constructed on the server at the time the document is requested. Dynamic documents have many advantages over static documents. For instance, the contents of the documents can be *tailor-made*, and *up-to-date*.

A natural extension of the dynamic-document model is the concept of *interactive* services, which is illustrated in Figure 11.1. Here the client does not

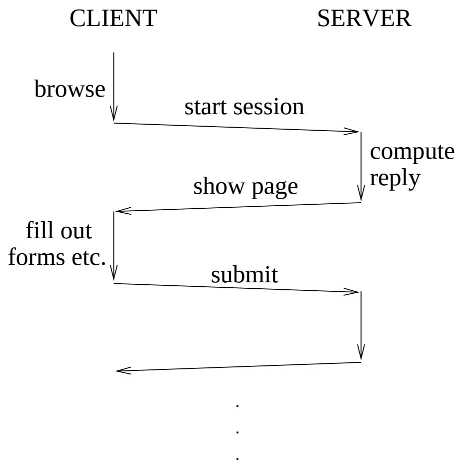


Figure 11.1: An interactive web session

browse a number of more or less independent statically or dynamically generated pages but is guided through a *session* controlled by a session thread on the server. This session can involve a number of user interactions. The session is initiated by the client submitting a “start session” request. The server then starts a thread controlling the new session. This thread generates a reply page which is sent back to the client. The page typically contains some input fields that are filled in by the client. That information is sent to the server, which then generates the next reply, and so on, until the session terminates.

This session concept allows a large class of services to be defined. However, a number of practical problems needs to be solved in order to implement this model on top of the CGI model.

11.2.2 CGI scripts and sequential session threads

As explained above, a web service session consists of a sequential computation that along the way presents information to the client and waits for replies. However, CGI is a state-less protocol, meaning that execution of a CGI script only lasts until a page is shown to the web client. This fact makes it rather tedious to program larger web services involving many client interactions. The sequential computation has to be split up into the small bits of computation that happen between client interactions. Each of these small bits will then constitute a CGI script or an instance of a CGI call.

Furthermore, to achieve persistency of the local state, one has to store and restore it explicitly between CGI-calls, for instance "hidden" in the web page sent to the client. For simple services where the full session approach is not needed this stateless-server approach might be preferable, but it is clearly inadequate in general.

Thus, the problem of forced termination of the CGI script at each client interaction is two-fold:

- Having to deal with many small scripts makes the *writing* and *maintenance* of a web service rather difficult because the control-flow of the service tends to become less clear from the program code.

- Starting up a whole new process every time a client interaction is performed is expensive in itself. On top of this a complete image of the local state has to be stored and restored each time a client interaction is required. The local state can potentially hold a lot of data, such as database contents. Thus one gets a *substantial overhead* in the execution of a web service.

We provide a simple solution which splits CGI scripts into two components, namely *connectors* and *session threads*. A connector is a tiny transient CGI script that redirects input to a session thread, receives the response from that thread, and redirects it back to the web client. The session threads are persistent processes running residently on the web server. They survive CGI calls and can therefore implement a long sequential computation involving several client interactions. The use of transient connectors and persistent session threads decreases the difficulty of writing and maintaining web services. Furthermore, it improves substantially on the overhead of the web server during execution of a service.

11.2.3 Other CGI shortcomings

Traditionally, reply pages from session threads are sent directly to the client. That is, the session thread (or the connector if using the system described above) writes the page to standard-output and the web server sends it on to the client browser. This basic approach imposes some annoying problems on the client:

- The client is not able to use "bookmarks" to identify the session, since selecting a bookmark might imply resending an old query to the server while the server expects a reply to a more recent interaction. It would be natural to the client if selecting a bookmarked session would continue the session from its current state. Obviously, this requires the server to always keep some kind of backup of the latest page sent to the client.
- In the session concept described in the previous section, it does not make sense to roll back execution of a session thread to a previous state. A thread can only be continued from its current point of execution. As a result of sending pages directly using the standard-output method, every new page shown to the client gets stacked up in the client's browser. This means that the stack of visited pages becomes filled up with references to outdated pages. One result is that the "back" button in the browser becomes rather useless.

We suggest a simple solution where—instead of sending the reply itself—the session thread writes its reply to a file visible to the client and then sends to the client a *reference* to the reply file. By choosing the same URL for the duration of the session, this reference can then function as an identification of that particular session. This solves both the problem with bookmarks and with the "back" button. Pressing "back" will now bring the client back to the web page where he started the session, which seems like a natural effect.

This method also opens up for an easy solution to another problem. Sometimes the server requires a long time to compute the information for the next page to be shown to the client. Naturally, the client may become *impatient* and lose interest in the service or assume that the server or the connection is down if no response is received within a certain amount of time. If confirmation in the form of a temporary response page is sent, the client will know that something is happening and that waiting will not be in vain.

This extra feature is implemented in the runtime system as follows. If a response is not ready within for instance 8 seconds, the connector responds with a reference to a temporary page (for instance saying "please wait") and terminates. This page will then automatically be loaded by the clients web browser and reload itself, say every 5 seconds. Once the session thread finishes its computation and the real response page is ready, the thread just replaces the temporary page with the real response page. This will have the effect that next time the page is reloaded, the real response page will be shown to the client.

This reloading can be done with standard HTML functionality. Of course the reloading causes some extra network traffic, but using this method is probably as close as one gets to server pushing in the world of CGI programming.

11.2.4 Handling safety requirements consistently

Another serious problem with traditional CGI programming is that concurrency control, such as synchronization of sessions and locking of shared variables, gets handled in an ad-hoc fashion. Typically, this is done using low-level semaphores supplied by the operating system.

As a result, web services often implement these aspects incorrectly resulting in unstable execution and sometimes even damaging behavior.

Our solution allows one to put safety requirements, such as mutual exclusion or much more complex requirements, separately in a centralized supervising process called the controller. This approach significantly simplifies the job of handling safety requirements. Also, since each of the requirements can be formulated separately, the solution is much more robust towards changes in various parts of the code.

It is generally considered inefficient and unsafe to have centralized components in distributed systems. However, in this case the bottleneck is more likely to be the HTTP/CGI server and the network than the safety controller. In spite of that, we do try to distribute the functionality of our safety controller as discussed in Section 11.5.

11.3 Components in the Runtime System

At any time there will be a number of *web clients* accessing the *HTTP/CGI server* through the CGI protocol. On the server side we will have a *controller* and a number of *session threads* running. The session threads access the global data and produce response pages for the web clients. From time to time a *connector* will be started as the result of a request from a web client. The connector will make contact with the running session thread. A connector is shut down again after having delegated the answer from a session thread back to the web client.

In the following we give a more detailed description of these components. For an overview of the components in the runtime system, see Figure 11.2.

Web clients Web clients are the users of the provided web service. They make use of the service essentially by filling in forms and submitting HTTP/CGI requests using a browser.

The HTTP/CGI server The HTTP/CGI server handles the incoming HTTP/CGI requests by retrieving web pages and starting up appropriate CGI scripts, in our case connectors. It also directs response pages back to the web clients.

Session threads Session threads are the resident processes running on the web server surviving several CGI calls. They represent the actual service code that implements the provided web service. They do calculations, search databases, produce response web pages, etc.

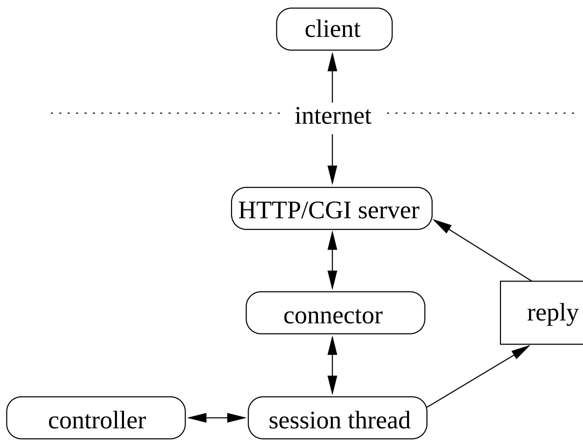


Figure 11.2: The runtime system

Connectors When a web client makes a request through the server, a connector is started up. If this request is the first one made, the controller starts up a new session thread corresponding to the request made by the web client. Otherwise—that is, if the web client wants to continue execution of a running session thread—the connector notifies the relevant session thread that a request has been made and forwards the input to that thread.

Reply pages Each session thread has a designated file which contains the current web page visible to the client of the session. When writing to this file, the whole contents is through a buffer updated atomically since the client may read the file at any time.

The controller The controller is a central component. It supervises session threads and has the possibility of suspending their execution at various points. This way it is ensured that the stated safety requirements are satisfied.

Furthermore, the runtime system also contains a *global-state database* (could be the file-system or a full-fledged database), and a *service manager*, which takes care of garbage-collecting abandoned session threads and other administrative issues.

11.4 Dynamics of the Runtime System

In this section we describe the dynamic behavior of the runtime system. We start by explaining the overall structure of the execution of a session thread. Starting from this, we present each of the possible thread transitions.

First, it is described how a session thread is started. Then, transitions involving interaction with a web client, that is, showing web pages and getting replies, are dealt with. Finally, the transitions involving interaction with the controller are presented.

For each transition we give a description of the components involved and their interaction.

11.4.1 Execution of a thread

The lifetime of a session thread is depicted in the diagram in Figure 11.3. When

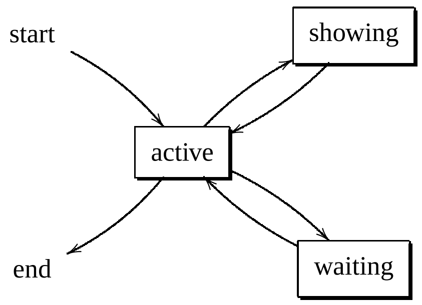


Figure 11.3: Possible states and transitions for a session thread

a thread is first started, it enters the state *active*. Here it can do all sorts of computations.

Eventually it reaches a point where it has composed a response HTML page. This page is shown to the web client and the thread enters the state *showing*. Here it waits for the web client to respond via yet another HTTP/CGI request. Upon re-submission the thread reenters the state *active* and resumes execution.

Note that in the world of naive CGI programming when moving from *active* to *showing* and back one would have to store a complete image of the local state before terminating the script. Then, when started again a new process would be started and the local state would have to be reconstructed from the image that was saved. This substantial overhead of saving and restoring local state is avoided completely by the use of transient connectors and resident threads.

While in state *active* a thread can get to a point in execution where safety critical computation, such as accessing a shared resource, needs to be carried out. When reaching such a point the thread asks the controller for permission to continue and enters the state *waiting*. When permission is granted from the controller the thread reenters the *active* state and continues execution.

With a traditional approach one would have to merge the code implementing the intricate details dealing with concurrency control with the service code. This intermixing would in addition to substantially reducing the readability of the code also increase the risk of introducing errors. Our solution separates the code dealing with concurrency control from the service code.

When the session is complete, the thread will leave the state *active* and end its execution.

11.4.2 Starting up a session thread

This section describes the transition from *start* to *active*.

When a new web client makes an HTTP/CGI request, the server will start up a new connector as a CGI script. Since this request is the first one made by the web client, a new thread is started according to the session name given in the request. As will be described later, a response page will be sent back to the client when the thread reaches a show call or a certain amount of time, for instance 8 seconds, has passed.

When a session thread is initiated or when it moves from *showing* to *active*, the contents of the reply file is immediately overwritten by a web page containing a "reply not ready—please wait" message and a "refresh" HTML command. The "refresh" command makes the browser reload the page every few seconds until the temporary reply file is overwritten by the real reply as described in the following section. The default contents of the "please wait" page can be overridden by the service programmer by simply overwriting the reply file with a message more appropriate for the specific situation.

11.4.3 Interaction with the client

During execution of a running thread the service can show a page to the web client and continue execution when receiving response from the client. In the following we describe these two actions.

Showing a page

This section describes the transition from *active* to *showing*.

During execution of a session thread one can do computations, inspect the input from the client, produce response documents, etc. When a response document has been constructed and the execution reaches a point where the page is to be shown to the client, the following actions will be taken:

1. First, the document to be shown is written to the reply file as indicated in Figure 11.2. This file always contains a "no cache" pragma-command, so that the client browser always fetches a new page even though the same URL is used for the duration of the whole session. Unfortunately we thereby lose the possibility of browser caching, but being restricted to building on top of existing standards we cannot get it all.

2. If the connector, that is, the CGI script started by the web client, has not already terminated due to the 8 second timeout, the session thread tells it that the reply page is ready. After this, the thread goes to sleep.

3. When the connector either has been waiting the 8 seconds or it receives the "reply ready" signal from the session thread, the connector writes a location-reference containing the URL for the reply page onto standard-output (using the CGI "location" feature), and then dies.

4. Finally, the HTTP/CGI server will transmit the URL back to the web clients browser which then will fetch the reply page through the HTTP/-CGI server and show it to the client.

In Figure 11.2, these actions describe a flow of data starting at the session thread and ending at the client.

Receiving client response

This section describes the transition from *showing* to *active*.

While the session thread is sleeping in the showing state, the web client will read the page, fill out appropriate form fields, and resubmit. This will result in the following flow of data from the client to the session thread (see Figure 11.2):

1. First, a request is made by the client via the CGI protocol. This request can be initiated either by clicking on a link or by pressing a submit button.

2. As a result, the HTTP/CGI server starts up a CGI script, that is, a connector.

3. The connector will then see that the client is already associated with a running thread and thus wake up that sleeping session thread and supply its new arguments.

11.4.4 Interaction with the controller

The controller allows the programmer to restrict the execution of a web service in such a way that stated safety requirements are satisfied.

Threads have built-in checkpoints at places where safety critical code is to be executed. At these checkpoints the thread must ask the controller for permission to continue. The controller, in turn, is constructed in such a way that it restricts execution according to the safety requirements and only allow threads that are not about to violate the requirements to continue.

In the following we describe in further detail the controller itself, what happens when session threads ask for permission, and how permission is granted by the controller.

The controller

The controller consists of three parts: some control logic, a number of checkpoint-event queues, and a timeout queue. Figure 11.4 gives an overview of the controller.

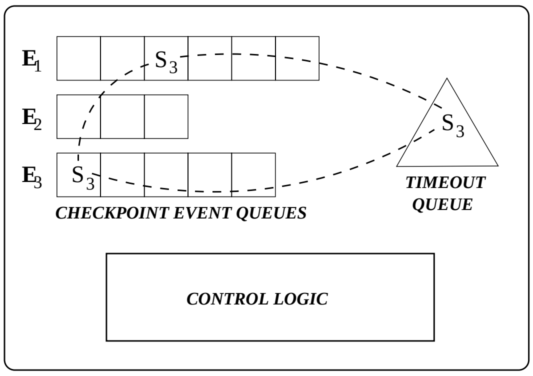


Figure 11.4: Components of the controller

The control logic The control logic is the actual component representing the safety requirements. It controls whether events are enabled, and hence when the various session threads may continue execution at checkpoints. One could imagine various approaches, such as, the use of finite state machines or petri-nets. For that reason, the internals of the control logic are not specified here. The only requirement is that the interface must contain the following two functions available to the runtime system:

- `check_enabled` — takes a checkpoint-event ID as argument and replies whether that event is currently enabled.
- `event_occurred` — takes the ID of an enabled checkpoint-event as argument and updates the internal state of control logic with the information that the event has occurred.

We explain in the following how these functions are used in the controller.

Checkpoint-event queues The *checkpoint-event queues* form the interface to the running threads of the service. There is a queue for each possible checkpoint event. When a thread reaches a checkpoint it asks the controller for permission to continue by adding its process-ID onto the queues corresponding to the events it wants to wait for at the checkpoint.

Timeout queue As an extra feature one can specify a *timeout* when asking the controller for permission to continue. For this purpose the controller has a timeout queue. If permission is not granted within the specified time bound, the controller wakes up the thread with the information that permission has not been granted yet, but a timeout event has occurred. The specified timeouts are put in the special timeout queue (which is implemented as a priority queue).

Asking for permission at checkpoints

This section describes the transition from *active* to *waiting*.

As mentioned earlier, one has the possibility of adding checkpoints to session code where critical code is to be executed. The runtime system interface makes some functions available to the service programmer for specifying checkpoints. Conceptually, the programmer uses them to specify a "checkpoint statement" as illustrated with an example in Figure 11.5. This example would have the effect that whenever a thread instance of this session reaches this point it will do the following:

1. First, it will tell the controller that it waits for either an $\mathtt{E}_1$ event, an $\mathtt{E}_3$ event, or a timeout of 20 seconds.
2. Having sent this request to the controller, the thread goes to sleep waiting for a response.

```
wait {
  case E1:
    ...
  case E3:
    ...
  timeout 20:
    ...
}
```

Figure 11.5: A checkpoint example

Controller actions

When the controller is up and running, it loops doing the following:

- If it receives a request to pass a checkpoint from a client, the controller pushes the ID of the client onto the appropriate queues. These entries are chained so that later, when permission is granted, they can all be removed at once. Figure 11.4 illustrates the effect of the example from Figure 11.5 where entries belonging to a session, $\mathtt{S}_3$, are in the $\mathtt{E}_1$, $\mathtt{E}_3$ and TIMEOUT queues.
- If a timeout has occurred, the controller deletes the affected entries in the queues and informs the involved thread.
- Otherwise, it will look for an enabled event using the `check_enabled` function from the control logic. If the queue corresponding to an enabled event is non-empty then the controller makes the event occur by doing the following:
  1. It removes the linked entries with the thread-ID of the enabled event from the respective queues,
  2. tells the control logic that the event has occurred using the `event_-occurred` function, and
  3. wakes up the involved thread with a "permission granted" signal containing the name of the event.

  If several events become enabled, a token-ring scheduling policy is used. This ensures fairness in the sense that if a thread waits for an enabled event, it will at some point be granted permission to continue.

Permission granted

This section describes the transition from *waiting* to *active*.

Having sent a request for permission to continue the thread is sleeping, waiting for the controller to make a response. If a "permission granted" signal is sent to the thread, it wakes up and continues, branching according to the event signaled by the controller. In the example checkpoint in Figure 11.5, if the controller grants permission for an `E`$_1$ event, execution is continued at the code following `case E`$_1$. If the controller sends a "timeout" signal, execution continues after `timeout`.

11.5 Extending the Runtime System

The runtime system described in the previous sections can be extended in several ways. The following extensions either have been implemented in an experimental version of the runtime system package or will be in near future. With these extensions, we believe that we begin reaching the limits of what is possible with the standard CGI protocol and the current functionality of standard browsers.

Distributed safety controller

To smoothen presentation, we have so far described the controller as one centralized component. In most cases it is possible to divide the control logic into independent parts controlling disjoint sets of checkpoint events. The controller can then be divided into a number of distributed control processes [71]. This way the problem of the controller being a bottleneck in the system is successfully avoided.

Service monitors

Using the idea of connectors and controllers, one can construct a "remote service monitor", that is, a program run by a super-client, which is able to access logs and statistics information generated by the connectors and controllers, and to inspect and change the global state and the state of the control logic in the controllers. This can be implemented by having a dedicated *monitor process* for each service.

Secure communication

The system presented here is quite vulnerable to hostile attacks. It is easy to hijack a session, since the URL of the reply file is enough to identify a session. A simple solution is to use random keys in the URLs, making it practically impossible to guess a session ID. Of course, all information sent between the clients browser and the server, such as the session ID and all data written in forms, can still be eavesdropped. To avoid this, we have been doing experiments with cryptography, making all communication completely secure in practice. This requires use of browser plug-ins, which unfortunately has not been standardized. The protocols being used in the experiments are RSA, DES3, and RIPE-MD160. They prevent hijacking, provide secure channels, and verify user ID—all transparently to the client.

Document clusters

In the session concept illustrated in Figure 11.1, only one page is generated and shown to the client at a time. However, often the service wants to generate a whole "cluster" of linked documents to the client and let the client browse these documents without involving the session thread. With the current implementation, a solution would be to program the possibility of browsing the cluster into the service code—inevitably a tedious and complicated task.

Document clusters can be implemented by simply having a reply file for each document in the cluster. Recall, however, that in the presented setup, the name of the reply file was fixed for the duration of a session. That way, the history buffer of the browser got a reasonable functionality. Therefore, to get that functionality we need a somewhat different approach: the reply files are not retrieved directly by the HTTP server but via a connector process. This connector receives the ID of the session thread in the CGI query string and the document number in a hidden variable.

Single process model

If all server processes (the session threads, safety controllers, etc.) are running on the same machine, that is, the possibility of distributing the processes is not being exploited, they might as well be combined into a single process using lightweight threads. This decreases the memory use (unless the operating system provides transparent sharing of code memory) and removes the overhead of process communication. The resulting system becomes something very close to being a dedicated web server. The important difference being that it still builds upon the CGI protocol.

11.6 Related Work

The idea of having persistent processes running residently on the server is central in the FastCGI [66] system. One difference is that FastCGI requires platform- and server-dependent support, while our approach works for all servers that support CGI. Also, our runtime system is tailored to support more specific needs.

A more detailed and formal description of how one can make use of safety requirements written separately in a suitable logic can be found in [71, 13]. A language for writing safety requirements is presented, the compilation process into a safety controller is described, and optimizations for memory usage and flow capacity of the controller are developed. A recent paper [45] generalizes these ideas resulting in a standard scheme for generating controllers for discrete event systems with both controllable and uncontrollable events.

The Mawl language [3, 29, 55] has been suggested as a domain-specific language for describing sequential transaction-oriented web applications. Its high-level notation is also compiled into low-level CGI scripts. Mawl directly provides programming constructs corresponding to global state, dynamic document, sessions, local state, imperative actions, and client interactions. This system shows great promise to facilitate the efficient production of reliable web services. While Mawl thus offers automatic synthesis of many advanced concepts, it still relies on standard low-level semaphore programming for concurrency control. Also, it does not have a FastCGI-like solution but in instead it is possible to compile a service into a dedicated server for that particular service. Though being faster than using simple CGI scripts this solution is, as opposed to using a FastCGI-like solution, not easily ported between different machine architectures.

11.7 Conclusions and Future Work

The implementation as briefly described in Appendix 11.7 constitutes the core of the `<bigwig>` tool which currently is being developed at BRICS. In the `<bigwig>` tool, the runtime system we propose here has shown to provide simple and efficient solutions to problems occurring more and more often due to the increased use of interactive web services. Furthermore, the session concept seems to constitute a framework which is very natural to use for designing complex services. By basing the design of the runtime system on very widely used protocols, the system is easy to incorporate. The further development of the runtime system can be followed on the `<bigwig>` homepage [74].

Implementation

A UNIX version of the runtime system has been implemented (in C) as a package "`runwig`" containing the following components (corresponding to Figure 11.2):

- The *connector*. It provides connection between the other components and the clients through the HTTP/CGI server.
- The *safety controller*, which handles syncronization and concurrency control. For the reasons described in Section 11.4.4, the control-logic is not included in the package but needs to be supplied separately.
- The *runtime library*, which is linked into the service code. It provides functions for easy interaction with the other components.

An experimental version of the runtime package implements the extensions described in Section 11.5. The `runwig` package—including all source code, detailed documentation, and examples—is available online[1].

`<bigwig>`

`<bigwig>` is a high-level programming language for developing interactive web services. Complete specifications are compiled into a conglomerate of lower-level technologies such as CGI-scripts, HTML, JavaScript, Java applets, and plug-ins running on top the runtime system presented in this paper. `<bigwig>`

[1]`http://www.brics.dk/bigwig/runwig/`

is an intellectual descendant of the Mawl project but is a completely new design and implementation with vastly expanded ambitions.

The `<bigwig>` language is really a collection of tiny domain-specific languages focusing on different aspects of interactive web services. To minimize the syntactic burdens, these contributing languages are held together by a C-like skeleton language. Thus, ¡bigwig¿ has the look and feel of C-programs with special data- and control-structures.

A `<bigwig>`service executes a dynamically varying number of threads. To provide a means of controlling the concurrent behavior, a thread may synchronize with a central controller that enforces the global behavior to conform to a regular language accepted by a finite-state automaton. That is, the 'control logic' in `<bigwig>` consists of finite-state automata. The controlling automaton is not given directly, but is computed (by the MONA [50, 62] system) from a collection of individual concurrency constraints phrased in first-order logic. Extensions with counters and negated alphabet symbols add expressiveness beyond regular languages.

HTML documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or attributes in tags. Such gaps may at runtime be plugged with concrete values. Since those values may themselves contain further gaps, this is a highly dynamic mechanism for building documents. The documents are represented in a very compressed format, and the plug operations takes constant time only. A flow-sensitive type checker ensures that documents are used in a consistent manner.

A standard service executes with hardly any security. Higher levels of security may be requested, such that all communications are digitally signed or encrypted using using 512 bit RSA and DES3. The required protocols are implemented using a combination of Java, Javascript, and native plug-ins.

The familiar struct and array datastructures are replaced with tuples and relations which allow for a simple construction of small relational databases. These are efficiently implemented and should be sufficient for databases no bigger than a few MBs (of which there are quite a lot). A relation may be declared to be external, which will automatically handle the connection to some external server. An external relation is accessed with (a subset of) the syntax for internal relations, which is then translated into SQL.

An important mechanism for gluing these components together is a fully general hygienic macro mechanism that allows ¡bigwig¿ programmers to extend the language by adding arbitrary new productions to its grammar. All nonterminals are potential arguments and result types for such macros that, unlike C-front macros, are soundly implemented with full alpha-conversions. Also, error messages remain sensible, since they are threaded back through macro expansion. This allows the definition of Very Domain-Specific Languages that contain specialized constructions for building chat rooms, shopping centers, and much more. Macros are also used to wrap concurrency constraints and other primitives in layers of user-friendly syntax.

Version 0.9 of `<bigwig>` is currently undergoing internal evaluation at BRICS. If you want to try it out, then contact us for more information. The documentation is very rough as yet, but this has a high priority in the next few months. The project is scheduled to deliver a version 1.0 of the `<bigwig>` tool in June 1999. This will be freely available in an open source distribution for UNIX.

Chapter 12

PowerForms: Declarative Client-Side Form Field Validation

with Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach

Abstract

All uses of HTML forms may benefit from validation of the specified input field values. Simple validation matches individual values against specified formats, while more advanced validation may involve interdependencies of form fields.

There is currently no standard for specifying or implementing such validation. Today, CGI programmers often use Perl libraries for simple server-side validation or program customized JavaScript solutions for client-side validation.

We present PowerForms, which is an add-on to HTML forms that allows a purely declarative specification of input formats and sophisticated interdependencies of form fields. While our work may be seen as inspiration for a future extension of HTML, it is also available for CGI programmers today through a preprocessor that translates a PowerForms document into a combination of standard HTML and JavaScript that works on all combinations of platforms and browsers.

The definitions of PowerForms formats are syntactically disjoint from the form itself, which allows a modular development where the form is perhaps automatically generated by other tools and the formats and interdependencies are added separately.

PowerForms has a clean semantics defined through a fixed-point process that resolves the interdependencies between all field values. Text fields are equipped with status icons (by default traffic lights) that continuously reflect the validity of the text that has been entered so far, thus providing immediate feed-back for the user. For other GUI components the available options are dynamically filtered to present only the allowed values.

PowerForms are integrated into the `<bigwig>` system for generating interactive Web services, but is also freely available in an Open Source distribution as a stand-alone package.

12.1 Introduction

We briefly review some relevant aspects of HTML forms. The CGI protocol enables Web services to receive input from clients through forms embedded in HTML pages. An HTML form is comprised of a number of input fields each prompting the client for information.

The visual rendering of an input field and how to enter the information it requests is determined by its type. The most widely used fields range from expecting lines of textual input to providing choices between a number of fixed options that were determined at the time the page was constructed. Many of the fields only differ in appearance and are indistinguishable to the server in the sense that they return the same kind of information. Fields of type `text` and `password`, although rendered differently, each expect one line of textual input from the client. Multiple lines of textual input can be handled through the `textarea` field. The fields of types `radio` and `select` both require exactly one choice between a number of static options, whereas an arbitrary number of choices are permitted by the `checkbox` and `select` (`multiple`) fields. Individual `radio` and `checkbox` fields with common name may be distributed about the form and constitute a group for which the selection requirements apply. The options of a `select` field, on the other hand, are grouped together in one place in the form. In addition, there are the more specialized fields, `image`, `file`, `button`, and `hidden`, which we shall not treat in detail. Finally, two fields control the behavior of the entire form, namely `reset` and `submit`, which respectively resets the form to its initial state and submits its contents to the server.

12.1.1 Input validation

Textual input fields could possibly hold anything. Usually, the client is expected to enter data of a particular form, for instance a number, a name, a ZIP-code, or an e-mail address. The most frequent solution is to determine on the server whether the submitted data has the required form, which is known as *server-side input validation*. If some data are invalid, then those parts are presented once again along with suitable error messages, allowing the client to make the necessary corrections. This process is repeated until all fields contain appropriate data. This solution is simple, but it has three well-known drawbacks:

- it takes time;
- it causes excess network traffic; and
- it requires explicit server-side programming.

Note that these drawbacks affect all parties involved. The client is clearly annoyed by the extra time incurred by the round-trip to the server for validation, the server by the extra network traffic and "wasted" cycles, and the programmer by the explicit programming necessary for implementing the actual validation and re-showing of the pages. An obvious solution to the first two drawbacks is

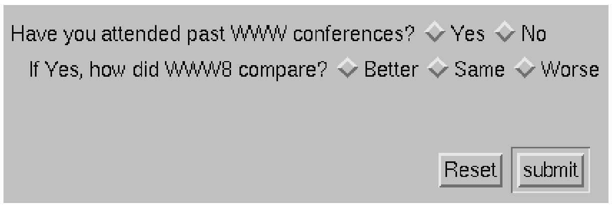

Have you attended past WWW conferences? ◇ Yes ◇ No
If Yes, how did WWW8 compare? ◇ Better ◇ Same ◇ Worse

Figure 12.1: Conference questionnaire.

to move the validation from the server to the client, yielding *client-side input validation*. The third drawback, however, is only partially alleviated. All the details of re-showing pages are no longer required, but the actual validation still needs to be programmed.

The move from server-side to client-side also opens for another important benefit, namely the possibility of performing the validation *incrementally*. The client no longer needs to click the submit button before getting the validation report. This allows errors to be be signalled as they occur, which clearly eases the task of correctly filling out the form.

12.1.2 Field interdependencies

Another aspect of validation involves interdependent fields. Many forms contain fields whose values may be constrained by values entered in other fields. Figure 12.1 exhibits a simple questionnaire from a conference, in which participants were invited to state whether they have attended past conferences and if so, how this one compared. The second question clearly depends on the first, since it may only be answered if the first answer was positive. Conversely, an answer to the second question may be required if the first answer was "Yes".

Such interdependencies are almost always handled on the server, even if the rest of the validation is addressed on the client-side. The reason is presumably that interdependencies require some tedious and delicate JavaScript code. This kind of validation is explicitly requested in the W3C working draft on extending forms [32]. One could easily imagine more advanced dependencies. Also, it would be useful if illegal selections could somehow automatically be deselected.

12.1.3 JavaScript programming

Traditionally, client-side input validation is implemented in JavaScript. We will argue that this may not be the best choice for most Web authors.

First of all, using a general-purpose programming language for a relatively specific purpose exposes the programmer to many unnecessary details and choices. A small high-level domain-specific language dedicated to input validation would involve only relevant concepts and thus be potentially easier to learn and use. Many assisting libraries exist [63], but must still be used in the context of a full programming language.

Secondly, JavaScript code has an operational form, forcing the programmer to think about the order in which the fields and their contents are validated.

However, the simplicity of the input validation task permits the use of a purely *declarative* approach. A declarative specification abstracts away operational details, making programs easier to read, write, and maintain. Also, such an approach is closer to composing HTML than writing JavaScript, making input validation available to more people. As stated in the W3C working draft on extending forms:

> "It should be possible to define a rich form, including validations, dependencies, and basic calculations without the use of a scripting language."

Our solution will precisely include such mechanisms for validations and dependencies.

Finally, the traditional implementation task is further complicated by diverging JavaScript implementations in various browsers. This forces the programmer to stay within the subset of JavaScript that is supported by all browsers—a subset that may be hard to identify. In fact, a number of sites and FAQs are dedicated to identifying this subset [94, 54]. A domain-specific language could be compiled into this common subset of JavaScript, implying that only the compiler writer will be concerned with this issue.

12.1.4 Our solution: PowerForms

As argued above, our solution is to introduce a high-level *declarative* and *domain-specific* language, called PowerForms, designed for incremental input validation.

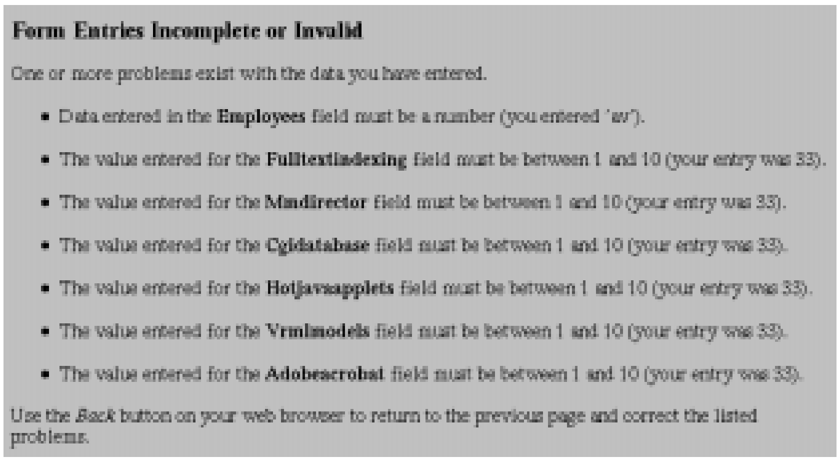
Section 12.2 presents our solution for simple validation; Section 12.3 extends this to handle field interdependencies; Section 12.4 exhibits how other common uses of JavaScript also can be handled through declarative specification; Section 12.5 presents the overall strategy of the translation to JavaScript; and Section 12.6 describes the availability of the PowerForms packages.

12.1.5 Related work

Authoring systems like Cold Fusion [21] can automate server-side verification of some simple formats, but even so the result is unsatisfactory. A typical response to invalid data is shown in Figure 12.2. It refers to the internal names of input fields which are unknown to the client, and the required corrections must be remembered when the form is displayed again.

Active Forms [86] is based on a special browser supporting Form Applets programmed as Tcl scripts. It does not offer high-level abstractions or integration with HTML.

Web Dynamic Forms [38] offer an ambitious and complex solution. They propose a completely new form model that is technically unrelated to HTML and exists entirely within a Java applet. Inside this applet, they allow complicated interaction patterns controlled through an event-based programming model in which common actions are provided directly and others may be programmed in Java. When a form is submitted, the data are extracted from



Form Entries Incomplete or Invalid

One or more problems exist with the data you have entered.

- Data entered in the **Employees** field must be a number (you entered 'av').
- The value entered for the **Fulltextindexing** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Mindirector** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Cgidatabase** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Hotjavaapplets** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Vrmlmodels** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Adobeacrobat** field must be between 1 and 10 (your entry was 33).

Use the *Back* button on your web browser to return to the previous page and correct the listed problems.

Figure 12.2: Typical server-side validation.

the applet and treated as ordinary HTML form data. The intervening years have shown that Web authors prefer to use standard HTML forms instead and then program advanced behavior in JavaScript. Thus, our simpler approach of automatically generating this JavaScript code remains relevant. An important reason to stay exclusively with HTML input fields is that they can be integrated into HTML tables to control their layout.

The XHTML-FML language [73] also provides a means for client-side input validation by adding an attribute called `ctype` to textual input fields. However, this attribute is restricted to a (large) set of predefined input validation types and there is no support for field inderdependency.

Our PowerForms notation is totally declarative and requires no programming skills. Furthermore, it is modular in the sense that validation can be added to an input field in an existing HTML form without knowing anything but its name. The validation markup being completely separate from the form markup allows the layout of a form to be redesigned at any time in any HTML editor.

12.2 Validation of Input Formats

The language is based on regular expressions embedded in HTML that is subsequently translated into a combination of standard HTML and JavaScript. This approach benefits from an efficient implementation through the use of finite-state automata which are interpreted by JavaScript code.

Named formats may be associated to fields whose values are then required to belong to the corresponding regular sets. The client is continuously receiving feedback, and the form can only be submitted when all formats are satisfied. The server should of course perform a double-check, since the JavaScript code is open to tampering.

Regular expressions denoting sets of strings are a simple and familiar formalism for specifying the allowed values of form fields. As we will demonstrate, all reasonable input formats can be captured in this manner. Also, the underlying technology of finite-state automata gives a simple and efficient implementation strategy.

12.2.1 Syntax

We define a rich XML syntax [20] for regular expressions on strings:

```
regexp → <const value=stringconst/> |
         <empty/> |
         <anychar/> |
         <anything/> |
         <charset value=stringconst/> |
         <fix low=intconst high=intconst/> |
         <relax low=intconst high=intconst/> |
         <range low=charconst high=charconst/> |
         <intersection> regexp* </intersection> |
         <concat> regexp* </concat> |
         <union> regexp* </union> |
         <star> regexp </star> |
         <plus> regexp </plus> |
         <optional> regexp </optional> |
         <repeat count=intconst> regexp </repeat>
         <repeat low=intconst high=intconst> regexp </repeat>
         <complement> regexp </complement> |
         <regexp exp=stringconst/> |
         <regexp id=stringconst> regexp </regexp> |
         <regexp idref=stringconst/> |
         <regexp uri=stringconst/> |
         <include uri=stringconst/>
```

Here, *regexp** denotes zero or more repetitions of *regexp*. The nonterminals *stringconst*, *intconst*, and *charconst* have the usual meanings.

Note that the verbose XML syntax also allows standard Perl syntax for regular expressions through the construct `<regexp exp=`*stringconst*`/>`. Our full syntax is however more general, since it includes intersection, general complementation, import mechanisms, and a richer set of primitive expressions.

A regular expression is associated with a form field through a declaration:

```
formatdecl → <format name=stringconst
                     help=stringconst
                     error=stringconst>
               regexp
             </format>
```

The value of the optional `help` attribute will appear in the status line of the browser when the field has focus; similarly, the value of the optional `error` attribute will appear if the field contains invalid data.

The format takes effect for a form field of type type `text`, `password`, `select`, `radio`, or `checkbox` whose name is the value of the `name` attribute. The need for input formats is perhaps only apparent for `text` and `password` fields, but we need the full generality later in Section 12.3.

12.2.2 Semantics of regular expressions

Each regular expression denotes an inductively defined set of strings. The `const` element denotes the singleton set containing its `value`. The `empty` element denotes the empty set. The `anychar` element denotes the set of all characters. The `anything` element denotes the set of all strings. The `charset` denotes the set of characters in its `value`. The `fix` element denotes the set of numerals from `low` to `high` all padded with leading zeros to have the same length as `high`. The `relax` element denotes the set of numerals from `low` to `high`. The `range` element denotes the set of singleton strings obtained from the characters `low` to `high`. The `intersection` element denotes the intersection of the sets denoted by its children. The `concat` element denotes the concatenation of the sets denoted by its children. The `union` element denotes the union of the sets denoted by its children. The `star` element denotes zero or more concatenations of the set denoted by its child. The `plus` element denotes one or more concatenations of the set denoted by its child. The `optional` element denotes the union of the set containing the empty string and the set denoted by its child. The `repeat` element with attribute `count` denotes a fixed power of the set denoted by its child. The `repeat` element with attributes `low` and `high` denotes the corresponding interval of powers of the set denoted by its child, where `low` defaults to zero and `high` to infinity. The `complement` element denotes the complement of the set denoted by its child. The `regexp` element with attribute `exp` denotes the set denotes by its attribute value interpreted as a standard Perl regular expression. The `regexp` element with attribute `id` denotes the same set as its child, but in addition names it by the value of `id`. The `regexp` element with attribute `idref` denotes the same set as the regular expression whose name is the value of `idref`. It is required that each `id` value is unique throughout the document and that each `idref` value matches some `id` value. The `regexp` element with attribute `uri` denotes the set recognized by a precompiled automaton. The `include` element performs a textual insertion of the document denoted by its `url` attribute.

12.2.3 Semantics of format declarations

The effect on a form field of a regular expression denoting the set S is defined as follows. For a `text` or `password` field, the effect is to decorate the field with one of four annotations:

- *green light*, if the current value is a member of S;
- *yellow light*, if the current value is a proper prefix of a member of S;
- *red light*, if the current value is not a prefix of a member of a non-empty S; or
- *n/a*, if S is the empty set.

The form cannot be submitted if it has a yellow or red light. The default annotations, which are placed immediately to the right of the field, are tiny icons inspired by traffic lights, but they can be customized with arbitrary images

	traffic	star	check	ok	blank
green light	🚦		✓	OK	
yellow light	🚦	★			
red light	🚦	★	✗		
n/a	NA			OK	

Figure 12.3: Different styles of status icons.

to obtain a different look and feel as indicated in Figure 12.3. Other annotations, like colorings of the input fields, would also seem reasonable, but current limitations in technology make this impossible.

For a `select` field, the effect is to filter the `option` elements, allowing only those whose values are members of S. There is a slight deficiency in the design of a singular `select`, since it in some browser implementations will always show one selected element. To account for the situation where no option is allowed, we introduce an extension of standard HTML, namely `<option value="foo" error>` which is legal irrespective of the format. The form cannot be submitted if the `error` option is selected, unless S is the empty set.

For a `radio` field, the effect is that the button can only be depressed if its value is a member of S; if S is not the empty set, then the form cannot be submitted unless one button is depressed. Note that the analogue of the `error` option is the case where no button is depressed.

For a `checkbox` field, the effect is that the button can only be depressed if its value is a member of S.

Using our mechanism, it is possible to create a *deadlocked* form that cannot be submitted. The simplest example is the following, assuming the input field below is the only one in the `radio` button group named `foo`:

```
<input type="radio" name="foo" value="aaa">
<format name="foo"><const value="bbb"></format>
```

Regardless of whether the radio button `foo` is depressed or not, `foo` will never satisfy its requirements. Thus, the form can never be submitted. This behavior exposes a flaw in the design of the form, rather than an inherent problem with our mechanisms.

12.2.4 Examples

All reasonable data formats can be expressed as regular expressions, some more complicated than others. A simple example is the password format for user

ID registration, seen in Figure 12.4, which is five or more characters not all alphabetic:

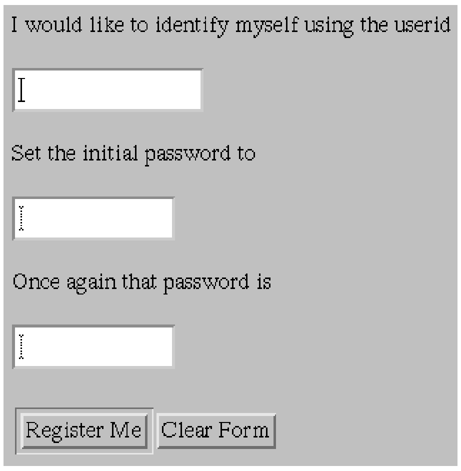
```
<regexp id="pwd">
  <intersection>
     <repeat low="5"><anychar/></repeat>
     <complement>
       <star>
         <union>
           <range low="a" high="z"/>
           <range low="A" high="Z"/>
         </union>
       </star>
     </complement>
  </intersection>
</regexp>
```

or alternatively using the Perl syntax where possible:

```
<regexp id="pwd">
  <intersection>
     <regexp exp=".{5,}"/>
     <complement>
       <regexp exp="[a-zA-Z]*"/>
     </complement>
  </intersection>
</regexp>
```

To enforce this format on the existing form, we just add the declarations:

```
<format name="Password1"><regexp idref="pwd"/></format>
<format name="Password2"><regexp idref="pwd"/></format>
```

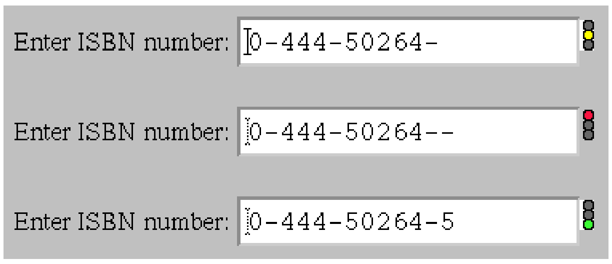

I would like to identify myself using the userid

Set the initial password to

Once again that password is

Figure 12.4: User ID registration.

At our Web site we show more advanced examples, such as legal dates including leap days, URIs, and time of day. As a final example, consider a simple format for ISBN numbers:


Enter ISBN number: 0-444-50264-

Enter ISBN number: 0-444-50264--

Enter ISBN number: 0-444-50264-5

Figure 12.5: Checking ISBN numbers.

```
<regexp id="isbn">
  <concat>
    <repeat count="9">
       <concat>
         <range low="0" high="9"/>
         <optional><charset value=" -"/></optional>
       </concat>
    </repeat>
    <charset value="0123456789X"/>
  </concat>
</regexp>
```

or more succinctly:

```
<regexp id="isbn">
  <regexp exp="([0-9]([ -]?)){9}[0-9X]"/>
</regexp>
```

An input field that exploits this format is:

```
Enter ISBN number: <input type=text name="isbn" size=20>
<format name="isbn"
        help="Enter an ISBN number"
        error="Illegal ISBN format">
  <regexp idref="isbn"/>
</format>
```

Initially, the field has a yellow light. This status persists, as seen in Figure 12.5, while we enter the text `"0-444-50264-"` which is a legal prefix of an ISBN number. Entering another `"-"` yields a red light. Deleting this character and entering `5` will finally give a legal value and a green light.

While the input field has focus, the `help` string appears in the status line of the browser. If the client attempts to submit the form with invalid data in this field, then the `error` text appears in an alert box.

An ISBN format that includes checksums can be described as a complex regular expression that yields a 201-state automaton. This full format would only accept `5` as the last digit, since that is the correct checksum. Such a regular expression could hardly be written by hand; in fact, we generated it using a C program. But as precompiled automata may be saved and provided as formats, this shows that our technology also allows us to construct and publish a collection of advanced default formats, similarly to the datatypes employed in XML Schema [12] and the predefined `ctype` formats suggested in [73].

12.3 Interdependencies of Form Fields

We present a simple, yet general mechanism for expressing interdependencies. We have strived to develop a purely declarative notation that requires no programming skills. Our proposal is based on dynamically evolving formats that are settled through a fixed-point process.

12.3.1 Syntax

We extend the syntax for formats as follows:

```
formatdecl  →  <format name=stringconst> format </format>

format      →  regexp |
               <if> boolexp
                  <then> format </then>
                  <else> format </else>
               </if> |
               <format id=stringconst> format </format> |
               <format idref=stringconst/>

boolexp     →  <match name=stringconst> regexp </match> |
               <equal name=stringconst value=stringconst/> |
               <and> boolexp* </and> |
               <or> boolexp* </or> |
               <not> boolexp* </not>
```

Now, the format that applies to a given field is dependent on the values of other fields. The specification is a binary decision tree, whose leaves are regular expressions and whose internal nodes are boolean expressions. Each boolean expression is a propositional combination of the primitive `match` and `equal` elements that each test the field indicated by `name`. Even this simple language is more advanced than required for most uses.

12.3.2 Semantics of boolean expressions

A boolean expression evaluates to true or false. For a `text` or `password` field, `equal` is true iff its current value equals `value`; `match` is true iff its current value is a member of the set denoted by `regexp`. For a `select` field, `equal` is true iff the value of a currently selected option equals `value`; `match` is true iff the value of a currently selected option is a member of the set denoted by `regexp`. For a collection of `radio` or `checkbox` fields, `equal` is true iff a button whose value equals `value` is currently depressed; `match` is true iff a button whose value is a member of the set denoted by `regexp` is currently depressed.

For the boolean operators, `and` is true iff all of its children are true, `or` is true if one of its children is true, and `not` is true if all of its children are false.

12.3.3 Semantics of interdependencies

Given a collection of form fields $F_1,\ldots,F_n$ with associated formats and values, we define an *iteration* which in order does the following for each F_i:

- evaluate the current format based on the current values of all form fields;
- update the field based on the new current format.

The updating varies with the type of the form field:

- for a `text` field, the status light is changed to reflect the relationship between the current value and the current format;
- for a `select` field, the options are filtered by the new format, and the selected options that are no longer allowed by the format are unselected; if the current selection of a singular `select` is disallowed, the `error` option is selected;
- for a `radio` or `checkbox` field, a depressed button is released if its value is no longer allowed by the format.

An iteration is *monotonic*, which intuitively means that it can only delete user data. Technically, an iteration is a monotonic function on a specific lattice of form status descriptions. It follows that repeated iteration will eventually reach a fixed-point. In fact, if b is the total number of `radio` and `checkbox` buttons, p is the total number of `select` options, and s is the number of singular `select`s, then at most $b + p + s + 1$ iterations are required. Usually, however, the fixed-point will stabilize after very few iterations; also, a compile-time dependency analysis can keep this number down. Only complex forms with a high degree of interdependency will require many iterations.

The behavior of a PowerForm is to iterate to a new fixed-point whenever the client changes an input field; furthermore, the form data can only be submitted when all the form fields are in a status that allows this.

Note that the fixed-point we obtain is dependent on the order in which the form fields are updated: permuting the fields may result in a different fixed-point. We choose to update the fields in the textual order in which they appear in the document. This is typically the order in which the client is supposed to consider them, and the resulting fixed-point appears to coincide with the intuitively expected behavior. For simpler forms, the order is usually not significant.

With form interdependency it is not only possible to create a deadlocked form that can never be submitted, but also to create buttons that can never be depressed. Consider again the example from Section 12.2. Since the value `aaa` is different from `bbb`, the `foo` button will instantly be released whenever it is depressed. Such behavior can of course also stem from more complicated interdependent behavior.

The possible behaviors of PowerForms can in principle be analyzed statically. Define the size $|R|$ of a regular expression to be the number of states in the corresponding minimal, deterministic finite-state automaton, and the size

$|F|$ of an input field to be the product of the sizes of all regular expressions that it may be tested against. Then a collection of input fields $F_1, \ldots, F_n$ determines a finite transition system with $|F_1||F_2|\cdots|F_n|$ states for which the reachability problem is decidable but hardly feasible in practice. We therefore leave it to the Web author to avoid aberrant behavior.

12.3.4 Examples

As a first example, we will redo the questionnaire from Figure 12.1:

```
Have you attended past WWW conferences?
<input type="radio" name="past" value="yes">Yes
<input type="radio" name="past" value="no">No
<br>
&nbsp;If Yes, how did WWW8 compare?
<input type="radio" name="compare" value="better">Better
<input type="radio" name="compare" value="same">Same
<input type="radio" name="compare" value="worse">Worse
```

To obtain the desired interdependence, we declare the following format:

```
<format name="compare">
   <if><equal name="past" value="yes"/>
     <then><complement><const value=""/></complement></then>
     <else><empty/></else>
   </if>
</format>
```

Only if the first question is answered in the positive, may the second group of radio buttons may be depressed and an answer is also required. A second example shows how radio buttons may filter the options in a selection:

```
Favorite letter group:
<input type="radio" name="group" value="vowel" checked>vowels
<input type="radio" name="group" value="consonant">consonants
<br>
Favorite letter:
<select name="letter">
  <option value="a">a
  <option value="b">b
  <option value="c">c
  ...
  <option value="x">x
  <option value="y">y
  <option value="z">z
</select>
```

The unadorned version of this form allows inconsistent choices such as `group` having value `vowel` and `letter` having value `z`. However, we can add the following format:

```
<format name="letter">
  <if><equal name="group" value="vowel"/>
    <then><charset value="aeiouy"/></then>
```

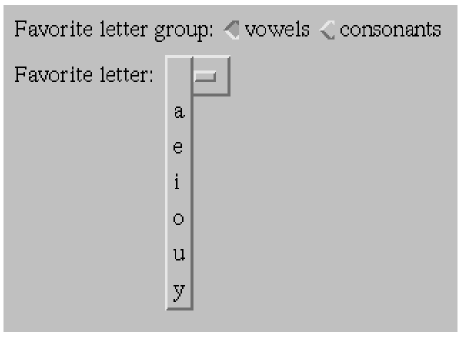


Figure 12.6: Only vowels are presented.

```
    <else><charset value="bcdfghjklmnpqrstvwxz"/></else>
  </if>
</format>
```

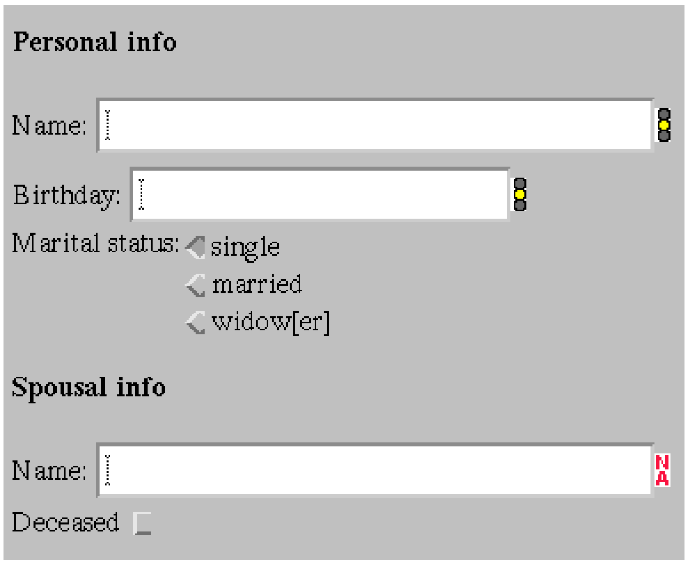
Apart from enforcing consistency, the induced behavior will make sure that the client is only presented with consistent options, as shown in Figure 12.6. Next, consider the form:

```
<b>Personal info</b>
<p>
Name: <input type="text" name="name" size="30"><br>
Birthday: <input type="text" name="birthday" size="20"><br>
<table border="0" cellpadding="0" cellspacing="0">
<tr><td valign="top">Marital status:</td>
<td><input type=radio name="marital" value="single" checked>single
<br>
<input type="radio" name="marital" value="married">married
<br>
<input type="radio" name="marital" value="widow">widow[er]
</td>
</tr>
</table>
<p>
<b>Spousal info</b>
<p>
Name: <input type="text" name="spouse" size="30"><br>
Deceased <input type="radio" name="deceased" value="deceased">
```

Several formats can be used here. For the birthday, we select from our standard library a 35-state automaton recognizing legal dates including leap days:

```
<format name="birthday">
  <regexp uri="http://www.brics.dk/bigwig/powerforms/date.dfa"/>
</format>
```

Among the other fields, there are some obvious interdependencies. Spousal info is only relevant if the `marital` status is not `single`, and the spouse can only be `deceased` if the `marital` status is `widow`:


Personal info

Name:

Birthday:

Marital status: single / married / widow[er]

Spousal info

Name:

Deceased

Figure 12.7: Collecting personal information.

```
<format name="spouse">
  <if><equal name="marital" value="married"/>
    <then><regexp idref="handle"/></then>
    <else>
      <if><equal name="marital" value="single"/>
        <then><empty/></then>
        <else><regexp idref="handle"/></else>
      </if>
    </else>
  </if>
</format>

<format name="deceased">
  <if><equal name="marital" value="widow"/>
    <then><const value="deceased"/></then>
    <else><empty/></else>
  </if>
</format>
```

Here, `handle` refers to some regular expression for the names of people. Note that if the `marital` status changes from `widow` to `single`, then the `deceased` button will automatically be released. Dually, it seems reasonable that after a change from `single` to `widow`, the `deceased` button should automatically be depressed. However, such action is generally not meaningful, since it may cause the form to oscillate between two settings. In our formalism, this would violate the monotonicity property that guarantees termination of the fixed-point iteration. Still, the form cannot be submitted until the `deceased` button is depressed for a `widow`. The initial form is shown in Figure 12.7.

An example of a more complex boolean expression involves the form in Figure 12.8. Here, simple formats determine that the correct style of phone

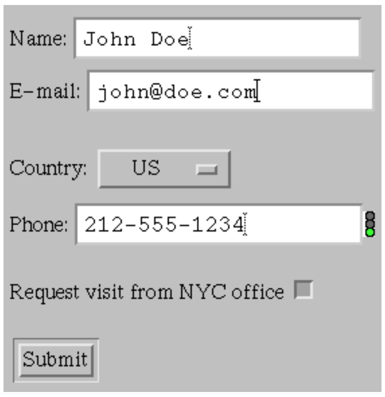


Figure 12.8: Collecting customer information.

numbers is used for the chosen country. The option of requesting a visit from the NYC office is only open to those customers who live in New York City. This constraint is enforced by the following format:

```
<format name="nyc">
  <if><and><equal name="country" value="US"/>
           <match name="phone">
              <concat>
                <union>
                  <const value="212"/>
                  <const value="347"/>
                  <const value="646"/>
                  <const value="718"/>
                  <const value="917"/>
                </union>
                <anything/>
              </concat>
           </match>
      </and>
     <then><anything/></then>
     <else><empty/></else>
  </if>
</format>
```

Residents from other cities will find that they cannot depress the button.

As a final example of the detailed control that we offer, consider the form in Figure 12.9 which invites users to request a new version of a product. Until the client has stated whether he has a license or not, it is impossible to choose a version. Once the choice has been made, licensed users can choose between all versions, others are limited to versions 1.1 and 1.2. The format on the last group of radio buttons is:

```
<format name="version">
  <if><equal name="license" value="yes"/>
      <then><anything/></then>
      <else>
```

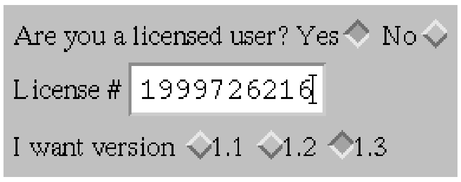


Figure 12.9: Collecting user information.

```
          <if><equal name="license" value="no"/>
              <then><union>
                      <const value="1.1"/>
                      <const value="1.2"/>
                    </union>
              </then>
              <else><empty/></else>
          </if>
        </else>
    </if>
  </format>
```

12.4 Applet results

Java applets can be used in conjunction with forms to implement new GUI components that collect data from the client. However, it is not obvious how to extract and validate data from an applet and submit it to the server on equal footing with ordinary form data.

We propose a simple mechanism for achieving this goal. We extend the applet syntax to allow `result` elements in addition to `param` elements. An example is the following:

```
<applet codebase="http://www.brics.dk/bigwig/powerapplets"
        code="slidebar.class">
   <param name="low" value="32">
   <param name="high" value="212">
   <result name="choice">
</applet>
```

When this applet is displayed, it shows a slide bar ranging over the interval [32..212]. When the form is submitted, the applet will be requested to supply a value for the `choice` result. This value is then assigned to a `hidden` form field named `choice` and will now appear with the rest of the form data. If the applet is not ready with the result, then the form cannot be submitted.

This extension only works for applets that are subclasses of the special class `PowerApplet` that we supply. It implements the method `putResult` that is used by the applet programmer to supply results, as well as the methods `resultsReady` and `getResult` that are called by the JavaScript code that implements the form submission.

In relation to PowerForms, applet results play the same role as input fields. Thus, they can have associated formats and be tested in boolean expressions. The value of an optional `error` attribute will appear in the alert box if an attempt is made to submit the form with a missing or invalid applet result.

12.5 Translation to JavaScript

A PowerForms document is parsed according to a very liberal HTML grammar that explicitly recognizes the special elements such as `format` and `regexp`. The generated HTML document retains most of the original structure, except that it contains the generated JavaScript code. Also, each input field is modified to include `onKeyup`, `onChange`, and `onClick` functions that react to modifications from the client.

A function `update_foo` is defined for each input field name `foo`. This function checks if the current data is valid and reacts accordingly. Another function `update_all` is responsible for computing the global fixed-point.

Each regular expression is by the compiler transformed into a minimal, deterministic finite-state automaton, which is directly represented in a JavaScript data structure. It is a simple matter to use an automaton for checking if a data value is valid. For `text` and `password` fields, the status lights green, yellow, and red correspond to respectively an accept state, a non-accept state, and the crash state. For efficiency, the generated automata are time-stamped and cached locally; thus, they are only recompiled when necessary.

The generated code is quite small, but relies on a 500 line standard library with functions for manipulating automata and the Document Object Model [1].

12.6 Availability

The PowerForms system is freely available in an open source distribution from our Web site located at `http://www.brics.dk/bigwig/powerforms/`. The package includes documentation, the examples from this paper and many more, and the compiler itself which is written in 4000 lines of C. The generated JavaScript code has been tested for Netscape on Unix and Windows and for Explorer on Windows.

PowerForms are also directly supported by the `<bigwig>` system which is a high-level language for generating interactive Web services [18, 16, 72, 71]. It is likewise available at `http://www.brics.dk/bigwig/`.

12.7 Conclusion

We have shown how to enrich HTML forms with simple, declarative concepts that capture advanced input validation and field interdependencies. Such forms are subsequently compiled into JavaScript and standard HTML. This allows the design of more complex and interesting forms while avoiding tedious and error-prone JavaScript programming.

We would like to thank the entire `<bigwig>` team for assisting in experiments with PowerForms. Thanks also goes to the PowerForms users, in particular Frederik Esser, for valuable feedback.

Chapter 13

Static Validation of Dynamically Generated HTML

with Anders Møller and Michael I. Schwartzbach

Abstract

We describe a static analysis of `<bigwig>` programs that efficiently decides if all dynamically computed XHTML documents presented to the client will validate according to the official DTD. We employ two data-flow analyses to construct a graph summarizing the possible documents. This graph is subsequently analyzed to determine validity of those documents. By evaluating the technique on a number of realistic benchmarks, we demonstrate that it is sufficiently fast and precise to be practically useful.

13.1 Introduction

Increasingly, HTML documents are dynamically generated by scripts running on a Web server, for instance using PHP, ASP, or Perl. This makes it much harder for authors to guarantee that such documents are really *valid*, meaning that they conform to the official DTD for HTML 4.01 or XHTML 1.0 [67]. Static HTML documents can easily be validated by tools made available by W3C and others. So far, the best possibility for a script author is to validate the dynamic HTML documents after they have been produced at runtime. However, this is an incomplete and costly process which does not provide any static guarantees about the behavior of the script. Alternatively, scripts may be restricted to use a collection of pre-validated templates, but this is generally not sufficiently expressive.

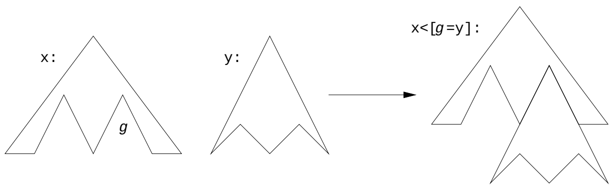
We present a novel technique for static validation of dynamic XHTML documents that are generated by a script. Our work takes place in the context of the `<bigwig>` language [18, 72], which is a full-fledged programming language for developing interactive Web services. In `<bigwig>`, XHTML documents are first-class citizens that are subjected to computations like all other data values. We instrument the compiler with an interprocedural data-flow analysis that extracts a grammatical structure, called a *summary graph*, covering the class of XHTML documents that a given program may produce. Based on this information, the compiler statically determines if all documents in the given class conform to the DTD for XHTML 1.0. To accomplish this, we need to reformulate DTDs in a novel way that may be interesting in its own right. The analysis has efficiently handled all available examples. Furthermore, our technique can be generalized to more powerful grammatical descriptions.

Outline

First, in Section 13.2, we give a brief introduction to dynamically generating XHTML documents in the `<bigwig>` language. Section 13.3 formally defines the notion of summary graphs. In Sections 13.4 and 13.5, the two parts of the data-flow analysis are specified. Then, in Section 13.6, a notion of abstract DTDs is defined and used for specifying XHTML 1.0. Section 13.7 describes the algorithm for validating summary graphs with respect to abstract DTDs. In Section 13.8 we evaluate our implementation on ten `<bigwig>` programs. Finally, in Sections 13.9 and 13.10 we briefly describe related techniques and plans and ideas for future work.

13.2 XHTML Documents in `<bigwig>`

XHTML documents are just XML trees. In the `<bigwig>` language, XML *templates* are first-class data values that may be passed and stored as any other values. Templates are more general than XML trees since they may contain *gaps*, which are named placeholders that can be *plugged* with templates and strings: If `x` is an XML template with a gap named *g* and `y` is another XML template or a text string, then the plug operation, `x<[`*g*`=y]`, results in a new template which is copy of `x` where a copy of `y` has been inserted into the *g* gap:



A `<bigwig>` service consists of a number of *sessions*. A session thread can be invoked by a client who is subsequently guided through a number of interactions, controlled by the service code on the server. A *document* is a template where all gaps have been filled. When a complete XHTML document has been built on the server, it can be shown to the client who fills in the input fields, selects menu options, etc., and then continues the session by submitting the input to the session thread.

This plug-and-show mechanism provides a very expressive way of dynamically constructing Web documents. It is described in more detail in [72, 18] where a thorough comparison with other mechanisms is given and other aspects of <bigwig> are described. Since templates can be plugged into templates, these are *higher-order* templates, as opposed to the less flexible templates in the Mawl language [55, 3, 4] where only strings can be plugged in.

Note that the number of gaps may both grow and shrink as the result of a plug operation. Also, gaps may appear in a non-local manner, as exemplified by the *what* gap being plugged with the template <b>BRICS</b> in the following simple example in the actual <bigwig> syntax:

```
service {
  html cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolor=[color]>
      <[contents]>
    </body>
  </html>;

  html greeting = <html>
    Hello <[who]>, welcome to <[what]>.
  </html>;

  html person = <html>
    <i>Stranger</i>
  </html>;

  session welcome() {
    html h;
    h = cover<[color="#9966ff",
               contents=greeting<[who=person]];
    show h<[what=<html><b>BRICS</b></html>];
  }
}
```

This service contains four constant templates and a session which when invoked will assemble a document using plug operations and show it to the client. Note that `color` is an *attribute gap* which can only be plugged with a string value, while the other gaps can also be plugged with templates. Constant templates are delimited by `<html>`...`</html>`. Implicitly, the mandatory surrounding `<html>` element is added to a document before being shown. Also, `<head>`, `<title>`, and `<body>` elements and a form with a default submit button is added if not already present. To simplify the presentation, we do not distinguish between HTML and XHTML since there are only minor syntactical differences. In the implementation, we allow HTML syntax but convert it to XHTML.

Note that <bigwig> is as general as all other languages for producing XML trees, since it is possible to define for each different element a tiny template like:

```
<html><ul style=[style]><[items]></ul></html>
```

that corresponds to a constructor function. The typical use of larger templates is mostly a convenience for the <bigwig> programmer.

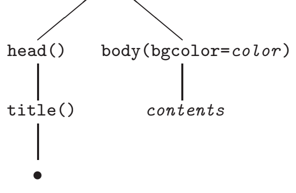
The `<bigwig>` compiler already contains an interprocedural data-flow analysis that keeps track of gaps and input fields in templates to enable type checking of plug and show operations [72]. That analysis statically ensures that the gaps are present when performing a plug operation and that the input fields in the documents being shown match the code that receives the values. However, the validity of the documents being shown has not been considered before, neither for `<bigwig>` or—to our knowledge—for any other programming language with such a flexible document construction mechanism.

XML Templates

We now formally define an abstract XML template. We are given an alphabet $\mathbf{\Sigma}$ of characters, an alphabet $\mathbf{E}$ of element names, an alphabet $\mathbf{A}$ of attribute names, an alphabet $\mathbf{G}$ of template gap names, and an alphabet $\mathbf{H}$ of attribute gap names. For simplicity, all alphabets are assumed to be disjoint. An *XML template* is generated by Φ in the following grammar:

$$\begin{array}{lll}
\Phi & \rightarrow \epsilon & \\
 & \rightarrow \bullet & \\
 & \rightarrow g & g \in \mathbf{G} \\
 & \rightarrow e(\Delta)\Phi & e \in \mathbf{E} \\
 & \rightarrow \Phi_1\Phi_2 & \\
\Delta & \rightarrow \epsilon & \\
 & \rightarrow (a = s) & a \in \mathbf{A},\ s \in \mathbf{\Sigma}^* \\
 & \rightarrow (a = h) & a \in \mathbf{A},\ h \in \mathbf{H} \\
 & \rightarrow \Delta_1\Delta_2 &
\end{array}$$

An XML template is a list of ordered trees where the internal nodes are *elements* with *attributes* and the leaves are either empty nodes, *character data* nodes, or *gap* nodes. Element attributes are generated by Δ. The $\bullet$ symbol represents an arbitrary sequence of character data. We ignore the actual data, since those are never constrained by DTDs, unlike attribute values which we accordingly represent explicitly. As an example, we view the `cover` template abstractly as follows if we ignore character data nodes consisting only of white-space:



We introduce a function:

$$gaps : (\Phi \cup \Delta) \rightarrow 2^{\mathbf{G} \cup \mathbf{H}}$$

which gives the set of gap names occurring in a template or attribute list:

$$\begin{array}{rcl} gaps(\epsilon) & = & \emptyset \\ gaps(\bullet) & = & \emptyset \\ gaps(g) & = & \{g\} \\ gaps(e(\delta)\phi) & = & gaps(\delta) \cup gaps(\phi) \\ gaps(\phi_1\phi_2) & = & gaps(\phi_1) \cup gaps(\phi_2) \\ gaps(a = s) & = & \emptyset \\ gaps(a = h) & = & \{h\} \\ gaps(\delta_1\delta_2) & = & gaps(\delta_1) \cup gaps(\delta_2) \end{array}$$

A template ϕ with a unique root element and with $gaps(\phi) = \emptyset$ is considered a complete *document*.

Programs

We represent a `<bigwig>` program abstractly as a control-flow graph with atomic statements at each program point. The actual syntax for `<bigwig>` is very liberal and resembles C or Java code with control structures and functions. For `<bigwig>` it is a simple task to extract the normalized representation. If the underlying language had a richer control structure, for instance with inheritance and virtual methods or higher-order functions, we would need a preliminary control-flow analysis to provide the control-flow graph.

A program uses a set X of XML template variables and a set Y of string variables. The atomic statements are:

x_i = x_j;	(template variable assignment)
x_i = ϕ;	(template constant assignment)
y_i = y_j;	(string variable assignment)
y_i = s;	(string constant assignment)
y_i = $\bullet$;	(arbitrary string assignment)
x_i = x_j<[g=x_k];	(template gap plugging)
x_i = x_j<[h=y_k];	(attribute gap plugging)
show x_i;	(client interaction)

where $x \in X$ and $y \in Y$ for each x and y. The assignments have the obvious semantics. The plug statement replaces all occurrences of a named gap with the given value. The `show` statement implicitly plugs all remaining gaps with ϵ before the template is displayed to the client. Also, the template is implicitly plugged into a wrapper template like the following:

```
<html>
  <head><title></title></head>
  <body>
    <form action="...">
      <[doc]>
      <input type="submit" value="continue">
    </form>
  </body>
</html>
```

for completing the document and adding a "continue" button. The `<head>`, `<title>`, `<body>`, and `<input>` elements are of course only added if not already present. Since we here ignore input fields in documents, the **receive** part of **show** statements is omitted in this description.

13.3 Summary Graphs

Given a program control-flow graph, we wish to extract a finite representation of all the templates that can possibly be constructed at runtime. A program contains a finite collection of constant XML templates that are identified through a mapping function:

$$\mathbf{f} : \mathbf{N} \to \Phi$$

where $\mathbf{N}$ is the finite set of indices of the templates occuring in the program. A program also contains a finite collection of string constants, which we shall denote by $\mathcal{C} \subseteq \mathbf{\Sigma}^*$. We now define a *summary graph* as a triple:

$$G = (R, E, \alpha)$$

where $R \subseteq \mathbf{N}$ is a set of *roots*, $E \subseteq \mathbf{N}\times\mathbf{G}\times\mathbf{N}$ is a set of *edges*, and $\alpha : \mathbf{N}\times\mathbf{H} \to \mathcal{S}$ is an attribute labeling function, where $\mathcal{S} = 2^{\mathcal{C}} \cup \{\bullet\}$. Intuitively, $\bullet$ denotes the set of all strings.

Each summary graph G defines a set of XML templates, which is called the *language* of G and is denoted $\mathcal{L}(G)$. Intuitively, this set is obtained by unfolding the graph from each root while performing all possible pluggings enabled by the edges and the labeling function. Formally, we define:

$$\mathcal{L}(G) = \{\phi \in \Phi \mid \exists r \in R : G, r \vdash \mathbf{f}(r) \Rightarrow \phi\}$$

where the derivation relation $\Rightarrow$ is defined for templates as:

$$\frac{}{G, n \vdash \epsilon \Rightarrow \epsilon} \qquad \frac{}{G, n \vdash \bullet \Rightarrow \bullet}$$

$$\frac{(n, g, m) \in E \quad G, m \vdash \mathbf{f}(m) \Rightarrow \phi}{G, n \vdash g \Rightarrow \phi}$$

$$\frac{G, n \vdash \delta \Rightarrow \delta' \quad G, n \vdash \phi \Rightarrow \phi'}{G, n \vdash e(\delta)\phi \Rightarrow e(\delta')\phi'}$$

$$\frac{G, n \vdash \phi_1 \Rightarrow \phi_1' \quad G, n \vdash \phi_2 \Rightarrow \phi_2'}{G, n \vdash \phi_1\phi_2 \Rightarrow \phi_1'\phi_2'}$$

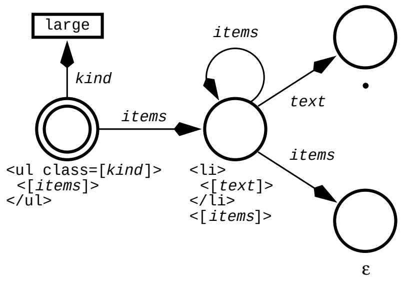
and for attribute lists as:

$$\frac{\alpha(n, h) \neq \bullet \quad s \in \alpha(n, h)}{G, n \vdash (a = h) \Rightarrow (a = s)}$$

$$\frac{\alpha(n, h) = \bullet \quad s \in \mathbf{\Sigma}^*}{G, n \vdash (a = h) \Rightarrow (a = s)}$$

$$\frac{G, n \vdash \delta_1 \Rightarrow \delta_1' \quad G, n \vdash \delta_2 \Rightarrow \delta_2'}{G, n \vdash \delta_1\delta_2 \Rightarrow \delta_1'\delta_2'}$$

As an example, consider the following summary graph consisting of four template nodes, four plug edges, and a single attribute labeling:



Template nodes, root nodes, and attribute labels are drawn as circles, double circles, and boxes, respectively. The language of this summary graph is the set of all `ul` lists of class `large` with one or more character data items.

13.4 Gap Track Analysis

To obtain sufficient precision of the actual validation analysis, we first perform an initial analysis that tracks the origins of gaps. We show in Section 13.5 exactly why this information is necessary.

Lattices

The lattice for this analysis is simply:

$$\mathcal{T} = (\mathbf{G} \cup \mathbf{H}) \to 2^{\mathbf{N}}$$

ordered by pointwise subset inclusion. For each program point ℓ we wish to compute an element of the derived lattice:

$$\mathit{TrackEnv}_\ell : X \to \mathcal{T}$$

which inherits its structure from $\mathcal{T}$. Intuitively, an element of this lattice tells us for a given variable x and a gap name g whether or not g can occur in the value of x, and if it can, which constant templates g can originate from.

Transfer Functions

Each atomic statement defines a transfer function $\mathit{TrackEnv}_\ell \to \mathit{TrackEnv}_\ell$ which models its semantics in a forward manner. If the argument is χ, then the results of applying this transfer function are:

x_i = x_j;	$\chi[x_i \mapsto \chi(x_j)]$
x_i = ϕ;	$\chi[x_i \mapsto \mathit{tfrag}(\phi, n)]$, where ϕ has index n
x_i = x_j<[g=x_k];	$\chi[x_i = \mathit{tplug}(\chi(x_j), g, \chi(x_k))]$
x_i = x_j<[h=y_k];	$\chi[x_i = \mathit{tplug}(\chi(x_j), h, \lambda p.\emptyset)]$

where we make use of some auxiliary functions:

$$tfrag(\phi, n) = \lambda p.\textsf{if } p \in gaps(\phi) \textsf{ then } \{n\} \textsf{ else } \emptyset$$

$$tplug(\tau_1, p, \tau_2) = \lambda q.\textsf{if } p{=}q \textsf{ then } \tau_2(q) \textsf{ else } \tau_1(q) \cup \tau_2(q)$$

For the remaining statement types, the transfer function is the identity function. The *tfrag* function states that all gaps in the given template originates from just that template. The *tplug* function adds all origins from the template being inserted and removes the existing origins for the gap being plugged.

The Analysis

It is easy to see that all transfer functions are monotonic, so we can compute the least fixed point iteratively in the usual manner [65]. The end result is for each program point ℓ an environment $track_\ell : X \to \mathcal{T}$, which we use in the following as a conservative, upper approximation of the origins of the gaps. We omit the proof of correctness.

13.5 Summary Graph Analysis

We wish to compute for every program point and for every variable a summary of its possible values. A set of XML templates is represented by a summary graph and a set of string values by an element of $\mathcal{S}$.

Lattices

To perform a standard data-flow analysis, we need both of these representations to be lattices. The set $\mathcal{S}$ is clearly a lattice, ordered by set inclusion and with $\bullet$ as a top element. The set of summary graphs, called $\mathcal{G}$, is also a lattice with the ordering defined by:

$$G_1 \sqsubseteq G_2 \quad \Leftrightarrow \quad R_1 \subseteq R_2 \ \wedge \ E_1 \subseteq E_2 \ \wedge \ \alpha_1 \sqsubseteq \alpha_2$$

where the ordering on $\mathcal{S}$ is lifted pointwise to labeling functions α. Clearly, both $\mathcal{S}$ and $\mathcal{G}$ are finite lattices. For each program point we wish to compute an element of the derived lattice:

$$Env_\ell = (X \to \mathcal{G}) \times (Y \to \mathcal{S})$$

which inherits its structure from the constituent lattices.

Transfer Functions

Each atomic statement defines a transfer function $Env_\ell \to Env_\ell$, which models its semantics. If the argument is the pair of functions (χ, γ) and ℓ is the entry program point of the statement, then the results are:

x_i = x_j;	$(\chi[x_i \mapsto \chi(x_j)], \gamma)$
x_i = ϕ;	$(\chi[x_i \mapsto frag(n)], \gamma)$, where ϕ has index n
y_i = y_j;	$(\chi, \gamma[y_i \mapsto \gamma(y_j)])$

`y`$_i$ `= `s`;`	$(\chi, \gamma[y_i \mapsto \{s\}])$
`y`$_i$ `= •;`	$(\chi, \gamma[y_i \mapsto \bullet])$
x_i `=` x_j`<[`g`=`x_k`];`	$(\chi[x_i \mapsto \mathit{gplug}(\chi(x_j), g, \chi(x_k), \mathit{track}_\ell(x_j))], \gamma)$
x_i `=` x_j`<[`h`=`y_k`];`	$(\chi[x_i \mapsto \mathit{hplug}(\chi(x_j), h, \gamma(y_k), \mathit{track}_\ell(x_j))], \gamma)$
`show` x_i`;`	(χ, γ)

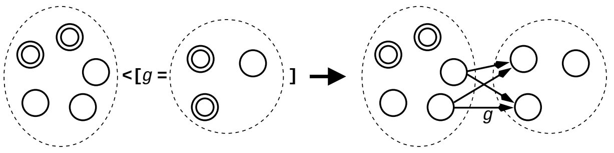
where we make use of some auxiliary functions:

$$\mathit{frag}(n) = (\{n\}, \emptyset, \lambda(m,h).\emptyset)$$

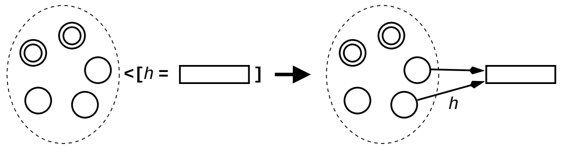
$$\begin{aligned}\mathit{gplug}(G_1, g, G_2, \tau) = (&R_1,\\ &E_1 \cup E_2 \cup\\ &\quad \{(n, g, m) \mid n \in \tau(g) \ \wedge\ m \in R_2\},\\ &\alpha_1 \sqcup \alpha_2)\end{aligned}$$

$$\begin{aligned}\mathit{hplug}(G, h, s, \tau) = (&R, E,\\ &\lambda(n, h').\text{if } n \in \tau(h) \text{ then } \alpha(n, h') \sqcup s\\ &\qquad\qquad\qquad\qquad \text{else } \alpha(n, h'))\end{aligned}$$

where $G_i = (R_i, E_i, \alpha_i)$ and $G = (R, E, \alpha)$. A careful inspection shows that all transfer functions are monotonic. The *frag* function constructs a tiny summary graph whose language contains only the given template. The *gplug* function joins the two summary graphs and adds edges from all relevant template gaps to the roots of the summary graph being inserted, which can be illustrated as follows:



The *hplug* function adds additional string values to the relevant attribute gaps:



We are now in a position to point out the need for the gap track analysis specified in Section 13.4. Without that initial analysis, the τ argument to *gplug* and *hplug* would always have to be the set $\mathbf{N}$ of all constant template indices to maintain soundness. Plugging a value into a gap g would then be modeled by adding an edge from all nodes having a g gap, even from nodes that originate from completely unrelated parts of the source code or nodes where the g gaps already have been filled. For instance, it is likely that a program building lists as in the summary graph example in Section 13.4 would contain other templates with a gap named `items`. Requiring each gap name to appear only in one constant template would solve the problem, but such a restriction would limit the flexibility of the document construction mechanism significantly. Hence, we rely on a program analysis to disregard the irrelevant nodes when adding plug edges.

The Analysis

Since we are working with monotonic functions on finite lattices, we can again use standard iterative techniques to compute a least fixed point [65]. The proof of soundness is omitted here, but it is similar to the one presented in [72]. The end result is for each program point ℓ an environment $summary_\ell : X \to \mathcal{G}$ such that $\mathcal{L}(summary_\ell(x_i))$ contains all possible XML templates that x_i may contain at ℓ. Those templates that are associated with `show` statements are required to validate with respect to the XHTML specification. We assume that the implicitly surrounding continue-button wrapper from Section 13.2 has been added already. Still, we must model the implicit plugging of empty templates and strings into the remaining gaps, so for the statement:

```
show x_i;
```

with entry program point q, the summary graph that must validate with respect to the XHTML DTD is:

$$close(summary_\ell(x_i), track_\ell(x_i))$$

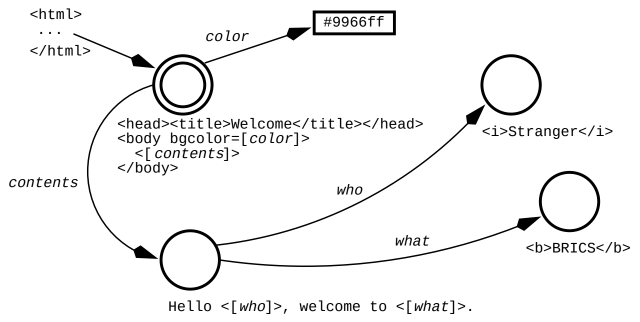
where *close* is defined by:

$$\begin{aligned} close(G,\tau) = (&R,\\ &E \cup \{(n,g,m_\epsilon)\,|\, n \in \tau(g)\},\\ &\lambda(n,h).\text{if } n \in \tau(h) \text{ then } \alpha(n,h) \sqcup \{\epsilon\}\\ &\qquad\qquad\qquad\quad \text{else } \alpha(n,h)) \end{aligned}$$

where $G = (R, E, \alpha)$ and it is assumed that $\mathbf{f}(m_\epsilon) = \epsilon$. The *close* function adds edges to an empty template for all remaining templates gaps, and adds the empty string as a possibility for all remaining attribute gaps.

The Example Revisited

For the small `<bigwig>` example in Section 13.2, the summary graph describing the document being shown to the client is inferred to be:



<html> ... </html>
color
#9966ff
<head><title>Welcome</title></head> <body bgcolor=[color]> <[contents]> </body>
<i>Stranger</i>
contents
who
what
<b>BRICS</b>
Hello <[who]>, welcome to <[what]>.

As expected for this simple case, the language of the summary graph contains exactly the single template actually being computed: Note that the XHTML template is implicitly completed with the `<html>` fragment.

13.6 An Abstract DTD for XHTML

XHTML 1.0 is described by an official DTD [67]. We use a more abstract formalism which is in some ways more restrictive and in others strictly more expressive. In any case, the DTD for XHTML 1.0 can be captured along with some restrictions that merely appear as comments in the official version. We define an abstract DTD to be a quintuple:

$$D = (\mathcal{N}, \rho, \mathcal{A}, \mathcal{E}, \mathcal{F})$$

where $\mathcal{N} \subseteq \mathbf{E}$ is a set of *declared* element names, $\rho \in \mathcal{N}$ is a *root* element name, $\mathcal{A} : \mathcal{N} \to 2^{\mathbf{A}}$ is an $\mathcal{N}$-indexed family of attribute name declarations, $\mathcal{E} : \mathcal{N} \to 2^{\mathcal{N}^\bullet}$ a family of element name declarations, and $\mathcal{F} : \mathcal{N} \to \Psi$ a family of formulas. We let $\mathcal{N}^\bullet = \mathcal{N} \cup \{\bullet\}$, where $\bullet$ represents arbitrary character data.

Intuitively, an abstract DTD consists of a number of element declarations whereof one is designated as the root. Each element declaration consists of an element name, a set of allowed attribute names, a set of allowed contents, and a formula constraining the use of the element with respect to its attribute values and contents. A formula has the syntax:

$$\begin{array}{lll}
\Psi & \to \Psi \wedge \Psi & \\
 & \to \Psi \vee \Psi & \\
 & \to \neg\Psi & \\
 & \to \mathit{true} & \\
 & \to \mathbf{attr}(a) & a \in \mathbf{A} \\
 & \to \mathbf{content}(c) & c \in \mathcal{N}^\bullet \\
 & \to \mathbf{order}(c_1, c_2) & c_i \in \mathcal{N}^\bullet \\
 & \to \mathbf{value}(a, \{s_1, \ldots, s_k\}) & a \in \mathbf{A},\ k \geq 1,\ s_i \in \mathbf{\Sigma}^*
\end{array}$$

We define the *language* of D as follows:

$$\mathcal{L}(D) = \{\rho(\delta)\phi \mid D \models \rho(\delta)\phi \ \wedge\ \mathit{gaps}(\phi) = \emptyset\}$$

That is, the language is the set of documents where the root element is ρ and the acceptance relation $\models$ is satisfied. This relation is defined inductively on templates as follows:

$$\frac{}{D \models \epsilon} \qquad \frac{}{D \models \bullet}$$

$$\frac{D \models \phi_1 \qquad D \models \phi_2}{D \models \phi_1\phi_2}$$

$$\frac{\begin{array}{cc} \mathit{names}(\delta) \subseteq \mathcal{A}(e) & D, \delta, \phi \models \mathcal{F}(e) \\ \mathit{set}(\phi) \subseteq \mathcal{E}(e) & D \models \phi \end{array}}{D \models e(\delta)\phi}$$

For each element, it is checked that its attributes and contents are declared and that the associated formula is satisfied. The auxiliary functions *names* and *set* are formally defined by:

$$\begin{array}{rcl} \mathit{names}(\epsilon) &=& \emptyset \\ \mathit{names}(a=s) &=& \{a\} \\ \mathit{names}(a=h) &=& \{a\} \\ \mathit{names}(\delta_1\delta_2) &=& \mathit{names}(\delta_1) \cup \mathit{names}(\delta_2) \end{array}$$

$$\begin{array}{rcl} \mathit{set}(\epsilon) &=& \emptyset \\ \mathit{set}(\bullet) &=& \{\bullet\} \\ \mathit{set}(g) &=& \emptyset \\ \mathit{set}(e(\delta)\phi) &=& \{e\} \\ \mathit{set}(\phi_1\phi_2) &=& \mathit{set}(\phi_1) \cup \mathit{set}(\phi_2) \end{array}$$

On formulas, the $\models$ relation is defined relative to the attributes and contents of an element:

$$\frac{D,\delta,\phi \models \psi_1 \quad D,\delta,\phi \models \psi_2}{D,\delta,\phi \models \psi_1 \wedge \psi_2}$$

$$\frac{D,\delta,\phi \models \psi_1}{\phi \models \psi_1 \vee \psi_2} \quad \frac{D,\delta,\phi \models \psi_2}{\phi \models \psi_1 \vee \psi_2}$$

$$\frac{}{D,\delta,\phi \models \mathit{true}} \quad \frac{D,\delta,\phi \not\models \psi}{D,\delta,\phi \models \neg\psi}$$

$$\frac{a \in \mathit{names}(\delta)}{D,\delta,\phi \models \mathbf{attr}(a)} \quad \frac{\mathit{exists}(\mathit{word}(\phi),c)}{D,\delta,\phi \models \mathbf{content}(c)}$$

$$\frac{\mathit{before}(\mathit{word}(\phi),c_1,c_2)}{D,\delta,\phi \models \mathbf{order}(c_1,c_2)}$$

$$\frac{a \notin \mathit{names}(\delta)}{D,\delta,\phi \models \mathbf{value}(a,\{s_1,\ldots,s_k\})}$$

$$\frac{(a,s_i) \in \mathit{atts}(\delta) \quad 1 \le i \le k}{D,\delta,\phi \models \mathbf{value}(a,\{s_1,\ldots,s_k\})}$$

The $\mathbf{attr}(a)$ formula checks whether an attribute of name a is present, and $\mathbf{content}(c)$ checks whether c occurs in the contents. The $\mathbf{value}(a,\{s_1,\ldots,s_k\})$ formula checks whether an a attribute has one of the values in $s_1,\ldots,s_k$ or is absent, and $\mathbf{order}(c_1,c_2)$ checks that no occurence of c_1 comes after an occurence of c_2 in the contents sequence. The auxiliary functions *atts* and *word* and the predicates *exists* and *before* are formally defined by:

$$\begin{array}{rcl} \mathit{atts}(\epsilon) &=& \emptyset \\ \mathit{atts}(a=s) &=& \{(a,s)\} \\ \mathit{atts}(a=h) &=& \{(a,h)\} \\ \mathit{atts}(\delta_1\delta_2) &=& \mathit{atts}(\delta_1) \cup \mathit{atts}(\delta_2) \end{array}$$

$$\begin{aligned}
\mathit{word}(\epsilon) &= \epsilon \\
\mathit{word}(\bullet) &= \bullet \\
\mathit{word}(g) &= \epsilon \\
\mathit{word}(e(\delta)\phi) &= e \\
\mathit{word}(\phi_1\phi_2) &= \mathit{word}(\phi_1)\mathit{word}(\phi_2)
\end{aligned}$$

$$\mathit{exists}(w_1\cdots w_k, c) \equiv \exists 1 \le i \le k : w_i = c$$

$$\begin{aligned}
\mathit{before}(w_1\cdots w_k, c_1, c_2) \equiv\ & \forall 1 \le i, j \le k : \\
& w_i = c_1 \wedge w_j = c_2 \Rightarrow i \le j
\end{aligned}$$

Two common abbreviations are $\textbf{unique}(c) \equiv \textbf{order}(c,c)$ ("c occurs at most once") and $\textbf{exclude}(c_1,c_2) \equiv \neg(\textbf{content}(c_1) \wedge \textbf{content}(c_2))$ ("c_1 and c_2 exclude each other").

Standard DTDs use restricted regular expressions to describe content sequences. Instead, we use boolean combinations of four basic predicates, each of which corresponds to a simple regular language. This is less expressive, since for example we cannot express that a content sequence must have exactly three occurrences of a given element. It is also, however, more expressive than DTDs since we allow the requirements on contents and attributes to be mixed in a formula. While the two formalism are thus theoretically incomparable, our experience is that XML languages described by DTDs or by more advanced schema languages typically are within the scope of our abstract notion.

Examples for XHTML

The DTD for XHTML 1.0 can easily be expressed in our formalism. The root element ρ is `html` and some examples of declarations and formulas are:

$$\begin{aligned}
\mathcal{A}(\texttt{html}) &= \{\texttt{xmlns}, \texttt{lang}, \texttt{xml:lang}, \texttt{dir}\} \\
\mathcal{E}(\texttt{html}) &= \{\texttt{head}, \texttt{body}\} \\
\mathcal{F}(\texttt{html}) &= \textbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \textbf{content}(\texttt{head}) \wedge \\
&\quad \textbf{content}(\texttt{body}) \wedge \textbf{unique}(\texttt{head}) \wedge \\
&\quad \textbf{unique}(\texttt{body}) \wedge \textbf{order}(\texttt{head}, \texttt{body})
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}(\texttt{head}) &= \{\texttt{lang}, \texttt{xml:lang}, \texttt{dir}, \texttt{profile}\} \\
\mathcal{E}(\texttt{head}) &= \{\texttt{script}, \texttt{style}, \texttt{meta}, \texttt{link}, \texttt{object}, \texttt{isindex} \\
&\quad \texttt{title}, \texttt{base}\} \\
\mathcal{F}(\texttt{head}) &= \textbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \textbf{content}(\texttt{title}) \wedge \\
&\quad \textbf{unique}(\texttt{title}) \wedge \textbf{unique}(\texttt{base})
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}(\texttt{input}) &= \{\texttt{id}, \texttt{class}, \texttt{style}, \texttt{title}, \texttt{lang}, \texttt{xml:lang}, \\
&\quad \texttt{dir}, \texttt{onclick}, \texttt{ondblclick}, \texttt{onmousedown}, \\
&\quad \texttt{onmouseup}, \texttt{onmouseover}, \texttt{onmousemove}, \\
&\quad \texttt{onmouseout}, \texttt{onkeypress}, \texttt{onkeydown}, \\
&\quad \texttt{onkeyup}, \texttt{type}, \texttt{name}, \texttt{value}, \texttt{checked}, \\
&\quad \texttt{disabled}, \texttt{readonly}, \texttt{size}, \texttt{maxlength}, \\
&\quad \texttt{src}, \texttt{alt}, \texttt{usemap}, \texttt{tabindex}, \texttt{accesskey}, \\
&\quad \texttt{onfocus}, \texttt{onblur}, \texttt{onselect}, \texttt{onchange}, \\
&\quad \texttt{accept}, \texttt{align}\} \\
\mathcal{E}(\texttt{input}) &= \emptyset \\
\mathcal{F}(\texttt{input}) &= \textbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge
\end{aligned}$$

$$\begin{aligned}
&\mathbf{value}(\texttt{checked}, \{\texttt{checked}\}) \wedge \\
&\mathbf{value}(\texttt{disabled}, \{\texttt{disabled}\}) \wedge \\
&\mathbf{value}(\texttt{readonly}, \{\texttt{readonly}\}) \wedge \\
&\mathbf{value}(\texttt{align}, \{\texttt{top}, \texttt{middle}, \texttt{bottom}, \\
&\qquad\qquad \texttt{left}, \texttt{right}\}) \wedge \\
&\mathbf{value}(\texttt{type}, \{\texttt{text}, \texttt{password}, \texttt{checkbox}, \\
&\qquad \texttt{radio}, \texttt{submit}, \texttt{reset}, \texttt{file}, \\
&\qquad \texttt{hidden}, \texttt{image}, \texttt{button}\}) \wedge \\
&(\mathbf{value}(\texttt{type}, \{\texttt{submit}, \texttt{reset}\}) \vee \mathbf{attr}(\texttt{name}))
\end{aligned}$$

In five instances we were able to express requirements that were only stated as comments in the official DTD, such as the last conjunct in $\mathcal{F}(\texttt{input})$. The full description of XHTML is available at `http://www.brics.dk/bigwig/xhtml/`.

Exceptions in `<bigwig>`

In one situation does `<bigwig>` allow non-standard XHTML notation. In the official DTD, the `ul` element is required to contain at least one `li` element. This is inconvenient, since the items of a list are often generated iteratively from a vector that may be empty. To facilitate this style of programming, `<bigwig>` allows empty `ul` elements but removes them at runtime before the XHTML is sent to the client. Accordingly, the abstract DTD that we employ differs from the official one in this respect. Similar exceptions are allowed for other kinds of lists and for tables. In the implementation, these fragment removal rules are specified the same way as the element constraints in the abstract DTD for XHTML, so essentially, we have just moved a few of the DTD constraints into a separate file.

13.7 Validating Summary Graphs

For every `show` statement, the data-flow analysis computes a summary graph $G = (R, E, \alpha)$. We must now for all such graphs decide the validation requirement:

$$\mathcal{L}(G) \subseteq \mathcal{L}(D)$$

for an abstract DTD $D = (\mathcal{N}, \rho, \mathcal{A}, \mathcal{E}, \mathcal{F})$. The root element name requirement of D is first checked separately by verifying that:

$$\forall r \in R: \ \mathbf{f}(r) = \rho(\delta)\phi \quad \text{for some } \delta \text{ and } \phi$$

Then for each sub-template $e(\delta)\phi$ of a template with index n in G we perform the following checks:

- $e \in \mathcal{N}$ (the element is defined)
- $\mathit{names}(\delta) \subseteq \mathcal{A}(e)$ (the attributes are declared)
- $\mathit{occurs}(n, \phi) \subseteq \mathcal{E}(e)$ (the content is declared)
- $n, \delta, \phi \Vdash \mathcal{F}(e)$ (the constraint is satisfied)

The validity relation $\Vdash$ is given by:[1]

$$\frac{n, \delta, \phi \Vdash \psi_1 \quad n, \delta, \phi \Vdash \psi_2}{n, \delta, \phi \Vdash \psi_1 \wedge \psi_2}$$

$$\frac{n, \delta, \phi \Vdash \psi_1}{n, \delta, \phi \Vdash \psi_1 \vee \psi_2} \qquad \frac{n, \delta, \phi \Vdash \psi_2}{n, \delta, \phi \Vdash \psi_1 \vee \psi_2}$$

$$\frac{}{n, \delta, \phi \Vdash \mathit{true}} \qquad \frac{n, \delta, \phi \nVdash \psi}{n, \delta, \phi \Vdash \neg\, \psi}$$

$$\frac{a \in \mathit{names}(\delta)}{n, \delta, \phi \Vdash \mathbf{attr}(a)} \qquad \frac{c \in \mathit{occurs}(n, \phi)}{n, \delta, \phi \Vdash \mathbf{content}(c)}$$

$$\frac{\mathit{order}(n, \phi, c_1, c_2)}{n, \delta, \phi \Vdash \mathbf{order}(c_1, c_2)}$$

$$\frac{a \notin \mathit{names}(\delta)}{n, \delta, \phi \Vdash \mathbf{value}(a, \{s_1, \ldots, s_k\})}$$

$$\frac{(a, s_i) \in \mathit{atts}(\delta) \quad 1 \le i \le k}{n, \delta, \phi \Vdash \mathbf{value}(a, \{s_1, \ldots, s_k\})}$$

$$\frac{(a, h) \in \mathit{atts}(\delta) \quad \alpha(n, h) \subseteq \{s_1, \ldots, s_k\}}{n, \delta, \phi \Vdash \mathbf{value}(a, \{s_1, \ldots, s_k\})}$$

where *occurs* is the least function satisfying:

$$\begin{array}{rcl}
\mathit{occurs}(n, \epsilon) & = & \emptyset \\
\mathit{occurs}(n, \bullet) & = & \{\bullet\} \\
\mathit{occurs}(n, g) & = & \displaystyle\bigcup_{(n,g,m) \in E} \mathit{occurs}(m, \mathbf{f}(m)) \\
\mathit{occurs}(n, e(\delta)\phi) & = & \{e\} \\
\mathit{occurs}(n, \phi_1\phi_2) & = & \mathit{occurs}(n, \phi_1) \cup \mathit{occurs}(n, \phi_2)
\end{array}$$

and *order* is the most restrictive function satisfying:

$$\begin{array}{rcl}
\mathit{order}(n, \epsilon, c_1, c_2) & = & \mathit{true} \\
\mathit{order}(n, \bullet, c_1, c_2) & = & \mathit{true} \\
\mathit{order}(n, g, c_1, c_2) & = & \displaystyle\bigwedge_{(n,g,m) \in E} \mathit{order}(m, \mathbf{f}(m), c_1, c_2) \\
\mathit{order}(n, e(\delta)\phi, c_1, c_2) & = & \mathit{true} \\
\mathit{order}(n, \phi_1\phi_2, c_1, c_2) & = & \mathit{order}(n, \phi_1, c_1, c_2) \wedge \\
 & & \mathit{order}(n, \phi_2, c_1, c_2) \wedge \\
 & & \neg\, (c_2 \in \mathit{occurs}(n, \phi_1) \wedge \\
 & & \quad c_1 \in \mathit{occurs}(n, \phi_2))
\end{array}$$

[1]Errata: Unfortunately, these rules are erroneous as presented. The rules can be amended by extending to a four-valued logic with the values *always*, *sometimes*, *never*, and *don't know*, with appropriate boolean connectives. The problem with using just *true* and *false* is that a predicate may hold only conditionally; the presence of negation implies that we cannot conservatively answer one over the other. This is for instance the case with the predicate `content` which may *sometimes* hold and *sometimes* not, depending of which of two templates is plugged into a gap.

The definition of the validity relation is straightforward. It duals the definition of the acceptance relation in Section 13.6, except that we now have to take gaps into account. Only the auxiliary functions, *occurs* and *order*, are non-trivial. The function $occurs(n, \phi)$ finds the subset of $\mathcal{N}^\bullet$ that can occur as contents of the current element after plugging some gaps according to the summary graph, and $order(n, \phi, c_1, c_2)$ checks that it is not possible to obtain an c_2 before an c_1 in the contents ϕ. These two functions are defined as fixed points because the summary graphs may contain loops. In the implementation we ensure termination by applying memoization to the numerous calls to *occurs* and *order*.

Note that the validation algorithm is both sound and complete[2] with respect to summary graphs: A graph is rejected if and only if its language contains a template that is not in the language of the abstract DTD. Thus, in the whole validation analysis the only source of imprecision is the data-flow analysis that constructs the summary graph.

Also note that our notion of abstract DTDs has a useful locality property: All requirements defined by an abstract DTD specify properties of single XML document nodes and their attributes and immidiate contents, so if some requirement is not fulfilled by a given summary graph, it is possible to give a precise error message.

13.8 Experiments

The validation analysis has been fully implemented as part of the `<bigwig>` system using a monovariant data-flow analysis framework. It has then been applied to all available benchmarks, some of which are shown in the following table:

Name	**Lines**	**Templates**	**Size**	**Shows**	**Time**
`chat`	65	3	(0,5)	2	0.1
`guess`	75	6	(0,3)	6	0.1
`calendar`	77	5	(8,6)	2	0.1
`xbiff`	561	18	(4,12)	15	0.1
`webboard`	1,132	37	(34,18)	25	0.6
`cdshop`	1,709	36	(6,23)	25	0.5
`jaoo`	1,941	73	(49,14)	17	2.4
`bachelor`	2,535	137	(146,64)	15	8.2
`courses`	4,465	57	(50,45)	17	1.3
`eatcs`	5,345	133	(35,18)	114	6.7

The entries for each benchmark are its name, the lines of code derived from a pretty print of the source with all macros expanded, the number of templates, the size $(|E|, |\alpha|)$ of the largest summary graph, the number of `show` statements, and the analysis time in seconds (on an 800 MHz Pentium III with Linux).

[2]Errata: Due to the issues mentioned in the previous footnote, the validation is sound but not complete. However, we have not encountered any spurious errors in practise, using the abstract DTD for XHTML.

The `chat` benchmark is a simple chat service, `guess` is a number guessing game, `calendar` shows a monthly calendar, `xbiff` is a soccer match reservation system, `webboard` is a bulletin board service, `cdshop` is a demonstration of an online shop, `jaoo` is a conference administration system, `bachelor` is a student management service, `courses` is a course administration system, and `eatcs` is a collection of services used by the EATCS organization. Some of the benchmarks are taken from the `<bigwig>` documentation, others are services currently being used or developed at BRICS.

The analysis found numerous validation errors in all benchmarks, which could then be fixed to yield flawless services. No false errors were reported. As seen in the table above, the enhanced compiler remains efficient and practical. The `bachelor` service constructs unusually complicated documents, which explains its high complexity.

Error Diagnostics

The `<bigwig>` compiler provides detailed diagnostic messages in case of validation errors. For the flawed example:

```
 1 service {
 2   html cover = <html>
 3     <head><title>Welcome</title></head>
 4     <body bgcolo=[color]>
 5       <table><[contents]></table>
 6     </body>
 7   </html>;
 8
 9    html greeting = <html>
10    <td>Hello <[who]>,<br clear=[clear]>
11         welcome to <[what]>.
12     </td>
13   </html>;
14
15   html person = <html>
16     <i>Stranger</i>
17   </html>;
18
19   session welcome() {
20     html h;
21     h = cover<[color="#9966ff",
22                contents=greeting<[who=person],
23                clear="righ"];
24     show h<[what=<html><b>BRICS</b></html>];
25   }
26 }
```

the compiler generates the following messages for the single **show** statement:

```
--- brics.wig:24: HTML validation:
brics.wig:4:
  warning: illegal attribute 'bgcolo' in 'body'
  template: <body bgcolo=[color]><form>...</form></body>
```

```
brics.wig:5:
  warning: possible illegal subelement ’td’ of ’table’
  template: <table><[contents]></table>
  contents: td
  plugs: contents:{brics.wig:22}

brics.wig:10:
  warning: possible element constraint violation at ’br’
  template: <br clear=[clear]/>
  constraint: value(clear,{left,all,right,clear,none})
  plugs: clear:{brics.wig:23}
```

At each error message, a line number of an XML element is printed together with an abbreviated form of the involved template, the names of the root elements of each template that can be plugged into the gaps, the constraint being violated, and the line numbers of the involved plug operations. Such reasonably precise error diagnostics is clearly useful for debugging.

13.9 Related Work

There are other languages for constructing XML documents that also consider validity. The XDuce language [43, 44] is a functional language in which XML templates are data types, with a constructor for each element name and pattern matching for deconstruction. A type is a regular expression over $\mathbf{E}^\bullet$. Type inference for pattern variables is supported. In comparison, we have a richer language and consequently need more expressive types that also describe the existence and capabilities of gaps. It seems unlikely that anything simpler than summary graphs would work. Also, we do not rely on type annotations. Since we perform an interprocedural data-flow analysis, we obtain a high degree of polymorphism that is difficult to express in a traditional type system. The XMλ language [60] compares similarly to our approach.

The initial design of the `<bigwig>` template mechanism was inspired by the Mawl language [55, 3, 4]. The main difference is that Mawl only allows strings to plugged into the gaps. Validating that Mawl programs only generate valid XHTML is therefore as easy as validating static documents, but such a simple document construction mechanism often becomes too restrictive for practical use. We have shown that using a highly flexible mechanism does not require validity guarantees to be sacrificed.

Most Web services are currently written either in Perl using CGI, in embedded scripting languages such as ASP, PHP, or JSP, or as server-integrated modules, for instance with Apache. Common to all these approaches is that there is no inherent type system for HTML or XML documents. In general, documents are constructed by concatenating text strings. These strings contain HTML or XML tags, attributes, etc., but the compiler or interpreter is completely unaware of that. This means that even *well-formedness*, that is, that tags are balanced and nested properly, which is one requirement for validity, becomes difficult to verify. We get that for free during parsing of the individual constant XML fragments and can concentrate on the many other validity requirements given by specific DTDs.

However, a common way of programming services in these languages is to use HTML or XML *constructor functions* to build documents more abstractly as trees instead of strings. This style is not enforced by the language, but if used consistently well-formedness is guaranteed. The difference between this and the `<bigwig>` style is that gaps in `<bigwig>` templates may appear non-locally, as described in Section 13.1, which gives a higher degree of flexibility. Since the constructor-based style is subsumed under the `<bigwig>` style as also described in Section 13.1, the summary graph technique could be applied for other languages.

13.10 Extensions and Future Work

Instead of our four basic predicates we could allow general regular expressions over the alphabet $\mathbf{E}^\bullet$. We could then still validate a summary graph, but this would reduce to deciding if a general context-free language is a subset of a regular language, which has an unwieldy algorithm compared to the simple transitive closures that we presently rely upon. Fortunately, our restricted regular languages appear sufficient. It is also possible to include many features from a richer XML schema language such as DSD [51], in particular context dependency and regular expression constraints on attribute values and character data.

Since our technique is parameterized in the choice of the abstract DTD, it easily generalizes to many other XML languages that can be described by such abstract DTDs. Finally, we could enrich `<bigwig>` with a set of operators for combining and deconstructing XML templates, making it a general XML transformation language. All such ideas readily permit analysis by means of summary graphs. However, a method for translating a DTD into a summary graph will be required.

13.11 Conclusion

We have combined a data-flow analysis with a generalized validation algorithm to enable the `<bigwig>` compiler to guarantee that all HTML or XHTML documents shown to the client are valid according to the official DTD. The analysis is efficient and does not generate many spurious error messages in practice. Furthermore, it provides precise error diagnostics in case a given program fails to verify.

Since our algorithm is parameterized with an abstract DTD, our technique generalizes in a straightforward manner to arbitrary XML languages that can be described by DTDs. In fact, we can even handle more expressive grammatical formalisms. The analysis has proved to be feasible for programs of realistic sizes. All this lends further support to the unique design of dynamic documents in the `<bigwig>` language.

Chapter 14

Language-Based Caching of Dynamically Generated HTML

with Anders Møller, Steffan Olesen, and Michael I. Schwartzbach

Abstract

Increasingly, HTML documents are dynamically generated by interactive Web services. To ensure that the client is presented with the newest versions of such documents it is customary to disable client caching causing a seemingly inevitable performance penalty. In the `<bigwig>` system, dynamic HTML documents are composed of higher-order templates that are plugged together to construct complete documents. We show how to exploit this feature to provide an automatic fine-grained caching of document templates, based on the service source code. A `<bigwig>` service transmits not the full HTML document but instead a compact JavaScript recipe for a client-side construction of the document based on a static collection of fragments that can be cached by the browser in the usual manner. We compare our approach with related techniques and demonstrate on a number of realistic benchmarks that the size of the transmitted data and the latency may be reduced significantly.

14.1 Introduction

One central aspect of the development of the World Wide Web during the last decade is the increasing use of *dynamically* generated documents, that is, HTML documents generated using e.g. CGI, ASP, or PHP by a server at the time of the request from a client [98, 8]. Originally, hypertext documents on the Web were considered to be principally *static*, which has influenced the design of protocols and implementations. For instance, an important technique for saving bandwidth, time, and clock-cycles is to cache documents on the client-side. Using the original HTTP protocol, a document that never or rarely changes can be associated an "expiration time" telling the browsers and proxy servers that there should be no need to reload the document from the server before that time. However, for dynamically generated documents that change on every request, this feature must be disabled—the expiration time is always set to "now", voiding the benefits of caching.

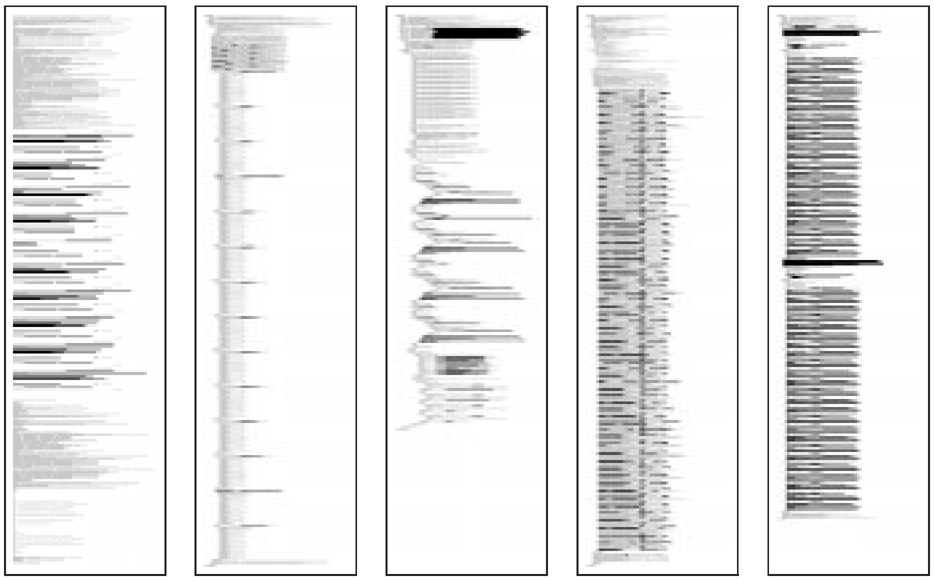
Even though most caching schemes consider all dynamically generated documents "non-cachable" [93, 9], a few proposals for attacking the problem have emerged [100, 69, 25, 47, 23, 31]. However, as described below, these proposals are typically not applicable for highly dynamic documents. They are often based on the assumptions that although a document is dynamically generated, 1) its construction on the server often does not have side-effects, for instance because the request is essentially a database lookup operation, 2) it is likely that many clients provide the same arguments for the request, or 3) the dynamics is limited to e.g. rotating banner ads. We take the next step by considering complex services where essentially every single document shown to a client is unique and its construction has side-effects on the server. A typical example of such a service is a Web-board where current discussion threads are displayed according to the preferences of each user. What we propose is not a whole new caching scheme requiring intrusive modifications to the Web architecture, but rather a technique for exploiting the caches already existing on the client-side in browsers, resembling the suggestions for future work in [98].

Though caching does not work for whole dynamically constructed HTML documents, most Web services construct HTML documents using some sort of constant templates that ideally ought to be cached, as also observed in [31, 97]. In Figure 14.1, we show a condensed view of five typical HTML pages generated by different `<bigwig>` Web services [18]. Each column depicts the dynamically generated raw HTML text output produced from interaction with each of our five benchmark Web services. Each non-space character has been colored either grey or black. The grey sections, which appear to constitute a significant part, are characters that originate from a large number of small, constant HTML templates in the source code; the black sections are dynamically computed strings of character data, specific to the particular interaction.

The `lycos` example simulates a search engine giving 10 results from the query "caching dynamic objects"; the `bachelor` service will based on a course roster generate a list of menus that students use to plan their studies; the `jaoo` service is part of a conference administration system and generates a graphical schedule of events; the `webboard` service generates a hierarchical list of active discussion threads; and the `dmodlog` service generates lists of participants in a course. Apart from the first simulation, all these examples are sampled from running services and use real data. The `dmodlog` example is dominated by string data dynamically retrieved from a database, as seen in Figure 14.1, and is thus included as a worst-case scenario for our technique. For the remaining four, the figure suggests a substantial potential gain from caching the grey parts.

The main idea of this paper is—automatically, based on the source code of Web services—to exploit this division into constant and dynamic parts in order to enable caching of the constant parts and provide an efficient transfer of the dynamic parts from the server to the client.

Using a technique based on JavaScript for shifting the actual HTML document construction from the server to the client, our contributions in this paper are:



(a) lycos (b) bachelor (c) jaoo (d) webboard (e) dmodlog

Figure 14.1: Benchmark services: cachable (grey) vs. dynamic (black) parts.

- an automatic characterization, based on the source code, of document fragments as *cachable* or *dynamic*, permitting the standard browser caches to have significant effect even on dynamically generated documents;
- a *compact representation* of the information sent to the client for constructing the HTML documents; and
- a generalization allowing a whole group of documents, called a *document cluster*, to be sent to the client in a single interaction and cached efficiently.

All this is possible and feasible due to the unique approach for dynamically constructing HTML documents used in the `<bigwig>` language [72, 18], which we use as a foundation. Our technique is non-intrusive in the sense that it builds only on preexisting technologies, such as HTTP and JavaScript—no special browser plug-ins, cache proxies, or server modules are employed, and no extra effort is required by the service programmer.

As a result, we obtain a simple and practically useful technique for saving network bandwidth and reviving the cache mechanism present in all modern Web browsers.

Outline

Section 14.2 covers relevant related work. In Section 14.3, we describe the `<bigwig>` approach to dynamic generation of Web documents in a high-level language using HTML templates. Section 14.4 describes how the actual document construction is shifted from server-side to client-side. In Section 14.5, we evaluate our technique by experimenting with five `<bigwig>` Web services. Finally, Section 14.6 contains plans and ideas for further improvements.

14.2 Related Work

Caching of dynamic contents has received increasing attention the last years since it became evident that traditional caching techniques were becoming insufficient. In the following we present a brief survey of existing techniques that are related to the one we suggest.

Most existing techniques labeled "dynamic document caching" are either server-based, e.g. [69, 25, 47, 100], or proxy-based, e.g. [23, 77]. Ours is client-based, as e.g. the HPP language [31].

The primary goal for server-based caching techniques is not to lower the network load or end-to-end latency as we aim for, but to relieve the server by memoizing the generated documents in order to avoid redundant computations. Such techniques are orthogonal to the one we propose. The server-based techniques work well for services where many documents have been computed before, while our technique works well for services where every document is unique. Presumably, many services are a mixture of the two kinds, so these different approaches might support each other well—however, we do not examine that claim in this paper.

In [69], the service programmer specifies simple cache invalidation rules instructing a server caching module that the request of some dynamic document will make other cached responses stale. The approach in [100] is a variant of this with a more expressive invalidation rule language, allowing classes of documents to be specified based on arguments, cookies, client IP address, etc. The technique in [47] instead provides a complete API for adding and removing documents from the cache. That efficient but rather low-level approach is in [25] extended with *object dependency graphs*, representing data dependencies between dynamic documents and underlying data. This allows cached documents to be invalidated automatically whenever certain parts of some database are modified. These graphs also allow representation of *fragments* of documents to be represented, as our technique does, but caching is not on the client-side. A related approach for caching in the Weave Web site specification system is described in [99].

In [77], a protocol for proxy-based caching is described. It resembles many of the server-based techniques by exploiting equivalences between requests. A notion of *partial request equivalence* allows similar but non-identical documents to be identified, such that the client quickly can be given an approximate response while the real response is being generated.

Active Cache [23] is a powerful technique for pushing computation to proxies, away from the server and closer to the client. Each document can be associated a *cache applet*, a piece of code that can be executed by the proxy. This applet is able to determine whether the document is stale and if so, how to refresh it. A document can be refreshed either the traditional way by asking the server or, in the other extreme, completely by the proxy without involving the server, or by some combination. This allows tailor-made caching policies to be made, and—compared to the server-side approaches—it saves network bandwidth. The drawbacks of this approach are: 1) it requires installation of new proxy servers which can be a serious impediment to wide-spread practical use, and 2) since there is no general automatic mechanism for characterizing document fragments as cachable or dynamic, it requires tedious and error-prone programming of the cache applets whenever non-standard caching policies are desired.

Common to the techniques from the literature mentioned above is that truly dynamic documents, whose construction on the server often have side-effects and essentially always are unique (but contain common constant fragments), either cannot be cached at all or require a costly extra effort by the programmer for explicitly programming the cache. Furthermore, the techniques either are inherently server-based, and hence do not decrease network load, or require installation of proxy servers.

Delta encoding [61] is based on the observation that most dynamically constructed documents have many fragments in common with earlier versions. Instead of transferring the complete document, a *delta* is computed representing the changes compared to some common base. Using a cache proxy, the full document is regenerated near the client. Compared to Active Cache, this approach is automatic. A drawback is—in addition to requiring specialized proxies—that it necessitates protocols for management of past versions. Such intrusions can obviously limit widespread use. Furthermore, it does not help with repetitions within a single document. Such repetitions occur naturally when dynamically generating lists and tables whose sizes are not statically known, which is common to many Web services that produce HTML from the contents of a database. Repetitions may involve both dynamic data from the database and static markup of the lists and tables.

The HPP language [31] is closely related to our approach. Both are based on the observation that dynamically constructed documents usually contain common constant fragments. HPP is an HTML extension which allows an explicit separation between static and dynamic parts of a dynamically generated document. The static parts of a document are collected in a *template* file while the dynamic parameters are in a separate *binding* file. The template file can contain simple instructions, akin to embedded scripting languages such as ASP, PHP, or JSP, specifying how to assemble the complete document. According to [31], this assembly and the caching of the templates can be done either using cache proxies or in the browser with Java applets or plug-ins, but it should be possible to use JavaScript instead, as we do.

An essential difference between HPP and our approach is that the HPP solution is not integrated with the programming language used to make the Web service. With some work it should be possible to combine HPP with popular embedded scripting languages, but the effort of explicitly programming the document construction remains. Our approach is based on the source language, meaning that all caching specifications are automatically extracted from the Web service source code by the compiler and the programmer is not required to be aware of caching aspects. Regarding cachability, HPP has the advantage that the instructions describing the structure of the resulting document are located in the template file which is cached, while in our solution the equivalent information is in the dynamic file. However, in HPP the constant fragments constituting a document are collected in a single template. This means that HTML fragments that are common to different document templates cannot be reused by the cache. Our solution is more fine-grained since it caches the individual fragments separately. Also, HPP templates are highly specialized and hence more difficult to modify and reuse for the programmer. Being fully automatic, our approach guarantees cache soundness. Analogously to optimizing compilers, we claim that the `<bigwig>` compiler generates caching code that is competitive to what a human HPP programmer could achieve. This claim is substantiated by the experiments in Section 14.5. Moreover, we claim that `<bigwig>` provides a more flexible, safe, and hence easier to use template mechanism than does HPP or any other embedded scripting language. The `<bigwig>` notion of *higher-order templates* is summarized in Section 14.3. A thorough comparison between various mechanisms supporting document templates can be found in [18].

As mentioned, we use compact JavaScript code to combine the cached and the dynamic fragments on the client-side. Alternatively, similar effects could be obtained using browser plug-ins or proxies, but implementation and installation would become more difficult. The HTTP 1.1 protocol [37] introduces both automatic compression using general-purpose algorithms, such as `gzip`, byte-range requests, and advanced cache-control directives. The compression features are essentially orthogonal to what we propose, as shown in Section 14.5. The byte-range and caching directives provide features reminiscent of our JavaScript code, but it would require special proxy servers or browser extensions to apply them to caching of dynamically constructed documents. Finally, we could have chosen Java instead of JavaScript, but JavaScript is more lightweight and is sufficient for our purposes.

14.3 Dynamic Documents in `<bigwig>`

The part of the `<bigwig>` Web service programming language that deals with dynamic construction of HTML documents is called DynDoc [72]. It is based on a notion of *templates* which are HTML fragments that may contain *gaps*. These gaps can at runtime be filled with other templates or text strings, yielding a highly flexible mechanism.

A `<bigwig>` *service* consists of a number of *sessions* which are essentially entry points with a sequential action that may be invoked by a client. When invoked, a session thread with its own local state is started for controlling the interactions with the client. Two built-in operations, *plug* and *show*, form the core of DynDoc. The *plug* operation is used for building documents. As illustrated in Figure 14.2, this operator takes two templates, `x` and `y`, and a gap name g and returns a copy of `x` where a copy of `y` has been inserted into every g gap. A template without gaps is considered a complete *document*. The *show* operation is used for interacting with the client, transmitting a given document

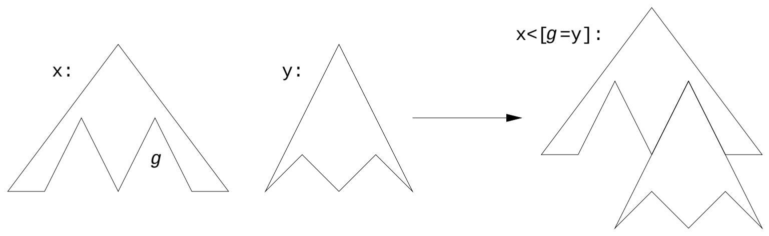


Figure 14.2: The *plug* operator.

to the client's browser. Execution of the client's session thread is suspended on the server until the client submits a reply. If the document contains input fields, the `show` statement must have a `receive` part for receiving the field values into program variables.

As in Mawl [55, 4], the use of templates permits programmer and designer tasks to be completely separated. However, our templates are *first-class* values in that they can be passed around and stored in variables as any other data type. Also they are *higher-order* in that templates can be plugged into templates. In contrast, Mawl templates cannot be stored in variables and only strings can be inserted into gaps. The higher-order nature of our mechanism makes it more flexible and expressive without compromising runtime safety because of two compile-time program analyses: a *gap-and-field analysis* [72] and an *HTML validation analysis* [17]. The former analysis guarantees that at every *plug*, the designated gap is actually present at runtime in the given template and at every *show*, there is always a valid correspondence between the input fields in the document being shown and the values being received. The latter analysis will guarantee that every document being shown is valid according to the HTML specification. The following variant of a well-known example illustrates the DynDoc concepts:

```
service {
  html ask = <html>What? <input name="what"></html>;
  html hello = <html>Hello, <b><[thing]></b>!</html>;

  session HelloWorld() {
    string s;
    show ask receive [s=what];
    hello = hello<[thing=s];
    show hello;
  }
}
```

Two HTML variables, `ask` and `hello`, are initialized with constant HTML templates, and a session `HelloWorld` is declared. The entities `<html>` and `</html>` are merely lexical delimiters and are not part of the actual templates. When invoked, the session first shows the `ask` template as a complete document to the client. All documents are implicitly wrapped into an `<html>` element and a form with a default "continue" button before being shown. The client fills out the *what* input field and submits a reply. The session resumes execution by storing the field value in the `s` variable. It then plugs that value into the *thing* gap of the `hello` template and sends the resulting document to the client. The following more elaborate example will be used throughout the remainder of the paper:

```
service {
  html cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolor=[color]>
      <[contents]>
    </body>
  </html>;

  html greeting = <html>
    Hello <[who]>, welcome to <[what]>.
  </html>;

  html person = <html><i>Stranger</i></html>;

  session welcome() {
    html h;
    h = cover<[color="#9966ff",
               contents=greeting<[who=person]];
    show h<[what=<html><b>BRICS</b></html>];
  }
}
```

It builds a "welcome to BRICS" document by plugging together four constant templates and a single text string, shows it to the client, and terminates. The higher-order template mechanism does not require documents to be assembled bottom-up: gaps may occur non-locally as for instance the *what* gap in `h` in the `show` statement that comes from the `greeting` template being plugged into the `cover` template in the preceding statement. Its existence is statically guaranteed by the gap-and-field analysis.

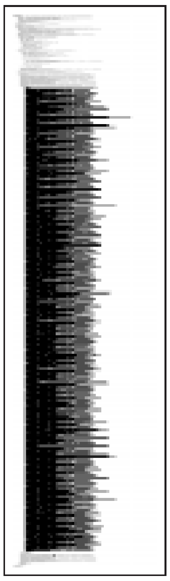
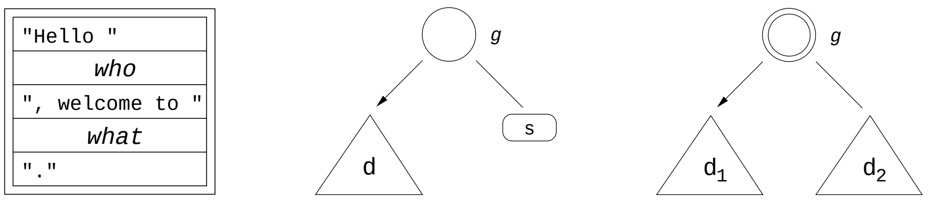


Figure 14.3: `webboard`

We will now illustrate how our higher-order templates are more expressive and provide better cachability compared to first-order template mechanisms. First note that ASP, PHP, and JSP also fit the first-order category as they conceptually correspond to having one single first-order template whose special code fragments are evaluated on the server and implicitly plugged into the template. Consider now the unbounded hierarchical list of messages in a typical Web bulletin board. This is easily expressed recursively using a small collection of DynDoc templates. However, it can never be captured by any first-order solution without casting from templates to strings and hence losing type safety. Of course, if one is willing to fix the length of the list explicitly in the template at compile-time,



(a) Leaf: `greeting` (b) Node: `strplug(d,`g`,s)` (c) Node: `plug(`d_1`,`g`,`d_2`)`

Figure 14.4: DynDocDag representation constituents.

it can be expressed, but not with unbounded lengths. In either case, sharing of repetitions in the HTML output is sacrificed, substantially cutting down the potential benefits of caching. Figure 14.3 shows the `webboard` benchmark as it would appear if it had been generated entirely using first-order templates: only the outermost template remains and the message list is produced by one big dynamic area. Thus, nearly everything is dynamic (black) compared to the higher-order version displayed in Figure 14.1(d).

Languages without a template mechanism, such as Perl and C, that simply generate documents using low-level `print`-like commands generally have too little structure of the output to be exploited for caching purposes.

All in all, we have with the *plug-and-show* mechanism in `<bigwig>` successfully transferred many of the advantages known from static documents to a dynamic context. The next step, of course, being caching.

14.3.1 Dynamic Document Representation

Dynamic documents in `<bigwig>` are at runtime represented by the *DynDocDag* data structure supporting four operations: constructing constant templates, `constant(c)`; string plugging, `strplug(d,`g`,s)`; template plugging, `plug(`d_1`,`g`,` d_2`)`; and showing documents, `show(d)`. This data structure represents a dynamic document as a binary DAG (Directed Acyclic Graph), where the leaves are either HTML templates or strings that have been plugged into the document and where the nodes represent pluggings that have constructed the document.

A constant template is represented as an ordered sequence of its text and gap constituents. For instance, the `greeting` template from the BRICS example service is represented as displayed in Figure 14.4(a) as a sequence containing two gap entries, *`who`* and *`what`*, and three text entries for the text around and between the gaps. A constant template is represented only *once* in memory and is shared among the documents it has been plugged into, causing the data structure to be a DAG in general and not a tree.

The string plug operation, `strplug`, combines a DAG and a constant string by adding a new string plug root node with the name of the gap, as illustrated in Figure 14.4(b). Analogously, the `plug` operation combines two DAGs as shown in Figure 14.4(c). For both operations, the left branch is the document containing the gap being plugged and the right branch is the value being plugged

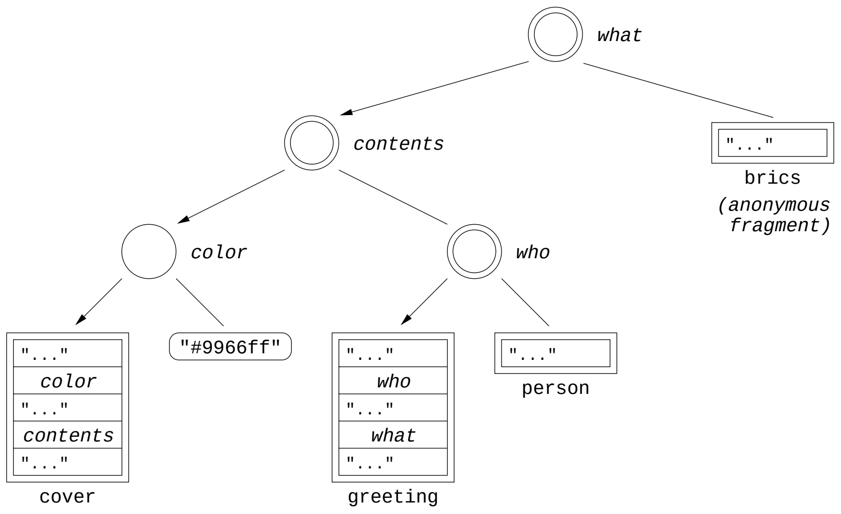


Figure 14.5: DynDocDag representation of the document shown in the BRICS example.

into the gap. Thus, the data structure merely records plug operations and defers the actual document construction to subsequent `show` operations.

Conceptually, the `show` operation is comprised of two phases: a *gap linking* phase that will insert a stack of links from gaps to templates and a *print traversal* phase that performs the actual printing by traversing all the gap links. The need for stacks comes from the template sharing.

The `strplug(d,`*g*`,s)`, `plug(`$\mathtt{d}_1$`,`*g*`,`$\mathtt{d}_2$`)`, and `show(d)` operations have optimal complexities, $O(1)$, $O(1)$, and $O(|\mathtt{d}|)$, respectively, where $|\mathtt{d}|$ is the lexical size of the `d` document.

Figure 14.5 shows the representation of the document shown in the BRICS example service. In this simple example, the DAG is a tree since each constant template is used only once. Note that for some documents, the representation is exponentially more succinct than the expanded document. This is for instance the case with the following recursive function:

```
html tree(int n) {
  html list = <html><ul><li><[gap]><li><[gap]></ul></html>;
  if (n==0) return <html>foo</html>;
  return list<[gap=tree(n-1)];
}
```

which, given n, in $O(n)$ time and space will produce a document of lexical size $O(2^n)$. This shows that regarding network load, it can be highly beneficial to transmit the DAG across the network instead of the resulting document, even if ignoring cache aspects.

14.4 Client-Side Caching

In this section we will show how to cache reoccurring parts of dynamically generated HTML documents and how to store the documents in a compact representation. The first step in this direction is to move the unfolding of the DynDocDag data structure from the server to the client. Instead of transmitting the unfolded HTML document, the server will now transmit a DynDocDag representation of the document in JavaScript along with a link to a file containing some generic JavaScript code that will interpret the representation and unfold the document on the client. Caching is then obtained by placing the constant templates in separate files that can be cached by the browser as any other files.

As we shall see in Section 14.5, both the caching and the compact representation substantially reduce the number of bytes transmitted from the server to the client. The compromise is of course the use of client clock cycles for the unfolding, but in a context of fast client machines and comparatively slow networks this is a sensible tradeoff. As explained earlier, the client-side unfolding is not a computationally expensive task, so the clients should not be too strained from this extra work, even with an interpreted language like JavaScript.

One drawback of our approach is that extra TCP connections are required for downloading the template files the first time, unless using the "keep connection alive" feature in HTTP 1.1. However, this is no worse than downloading a document with many images. Our experiments show that the number of transmissions per interaction is limited, so this does not appear to be a practical problem.

14.4.1 Caching

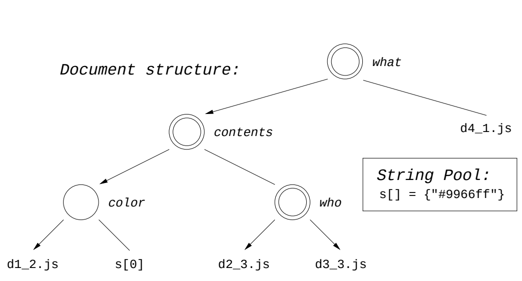
The DynDocDag representation has a useful property: it explicitly maintains a separation of the *constant templates* occurring in a document, the *strings* that are plugged into the document, and the *structure* describing how to assemble the document. In Figure 14.5, these constituents are depicted as framed rectangles, oval rectangles, and circles, respectively.

Experiments suggest that templates tend to occur again and again in documents shown to a client across the lifetime of a `<bigwig>` service, either because they occur 1) many times in the same document, 2) in many different documents, or 3) simply in documents that are shown many times. The strings and the structure parts, however, are typically dynamically generated and thus change with each document.

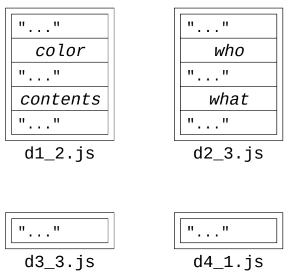
The templates account for a large portion of the expanded documents. This is substantiated by Figure 14.1, as earlier explained. Consequently, it would be useful to somehow cache the templates in the browser and to transmit only the dynamic parts, namely the strings and the structure at each `show` statement. This separation of cachable and dynamic parts is for the BRICS example illustrated in Figure 14.6.

As already mentioned, the solution is to place each template in its own file and include a link to it in the document sent to the client. This way, the caching mechanism in the browser will ensure that templates already seen are not retransmitted.

The first time a service shows a document to a client, the browser will obviously not have cached any of the JavaScript template files, but as more and



(a) Dynamic document structure reply file.



(b) Cachable template files.

Figure 14.6: Separation into cachable and dynamic parts.

more documents are shown, the client will download fewer and fewer of these files. With enough interactions, the client reaches a point of *asymptotic caching* where all constant templates have been cached and thus only the dynamic parts are downloaded.

Since the templates are statically known at compile-time, the compiler enumerates the templates and for each of them generates a file containing the corresponding JavaScript code. By postfixing template numbers with version numbers, caching can be enabled across recompilations where only some templates have been modified.

In contrast to HPP, our approach is entirely automatic. The distinction between static and dynamic parts and the DynDocDag structure are identified by the compiler, so the `<bigwig>` programmer gets the benefits of client-side caching without tedious and error-prone manual programming of bindings describing the dynamics.

14.4.2 Compact Representation

In the following we show how to encode the cachable template files and the reply documents containing the document representation. Since the reply documents are transmitted at each `show` statement, their sizes should be small. Decompression has to be conducted by JavaScript interpreted in browsers, so we do not apply general purpose compression techniques. Instead we exploit the inherent structure of the reply documents to obtain a lightweight solution: a simple yet compact JavaScript representation of the string and structure parts that can be encoded and decoded efficiently.

Constant Templates

A constant template is placed in its own file for caching and is encoded as a call to a JavaScript constructor function, `F`, that takes the number and version of the template followed by an array of text and gap constituents respectively constructed via calls to the JavaScript constructor functions `T` and `G`. For instance, the `greeting` template from the BRICS example gets encoded as follows:

```
F(T('Hello '),G(3),T(', welcome to '),G(4),T('.'));
```

Assuming this is version 3 of template number 2, it is placed in a file called `d2_3.js`. The gap identifiers *who* and *what* have been replaced by the numbers 3 and 4, respectively, abstracting away the identifier names. Note that such a file needs only ever be downloaded once by a given client, and it can be reused every time this template occurs in a document.

Dynamics

The JavaScript reply files transmitted at each `show` contain three document specific parts: *include directives* for loading the cachable JavaScript template files, the *dynamic structure* showing how to assemble the document, and a *string pool* containing the strings used in the document.

The structure part of the representation is encoded as a JavaScript string constant, by a uuencode-like scheme which is tuned to the kinds of DAGs that occur in the observed benchmarks.

Empirical analyses have exposed three interesting characteristics of the strings used in a document: 1) they are all relatively short, 2) some occur many times, and 3) many seem to be URLs and have common prefixes. Since the strings are quite short, placing them in individual files to be cached would drown in transmission overhead. For reasons of security, we do not want to bundle up all the strings in cachable string pool files. This along with the multiple occurrences suggests that we collect the strings from a given document in a string pool which is inlined in the reply file sent to the client. String occurrences within the document are thus designated by their offsets into this pool. Finally, the common prefix sharing suggests that we collect all strings in a *trie* which precisely yields sharing of common prefixes. As an example, the following four strings:

```
"foo",
"http://www.brics.dk/bigwig/",
"http://www.brics.dk/bigwig/misc/gifs/bg.gif",
"http://www.brics.dk/bigwig/misc/gifs/bigwig.gif"
```

are linearized and represented as follows:

```
"foo|http://www.brics.dk/bigwig/[misc/gifs/b(igwig.gif|g.gif)]"
```

When applying the trie encoding to the string data of the benchmarks, we observe a reduction ranging from 1780 to 1212 bytes (on `bachelor`) to 27728 to 10421 bytes (on `dmodlog`).

The reply document transmitted to the client at the `show` statement in the BRICS example looks like:

```
<html>
 <head>
  <script src="http://www.brics.dk/bigwig/dyndoc.js"></script>
  <script>I(1,2,3,4, 2,3,3,1);</script>
```

```
    <script>S("#9966ff");  D("/&Ë$Î&I%",2,8,4);</script>
   </head>
   <body onload="E();"></body>
  </html>
```

The document starts by including a generic 15K JavaScript library, `dyndoc.js`, for unfolding the DynDocDag representation. This file is shared among all services and is thus only ever downloaded once by each client as it is cached after the first service interaction. For this reason, we have not put effort into writing it compactly. The include directives are encoded as calls to the function `I` whose argument is an array designating the template files that are to be included in the document along with their version numbers. The `S` constructor function reconstructs the string trie which in our example contains the only string plugged into the document, namely “`#9966ff`”. As expected, the document structure part, which is reconstructed by the `D` constructor function, is not humanly readable as it uses the extended ASCII set to encode the dynamic structure. The last three arguments to `D` recount how many bytes are used in the encoding of a node, the number of templates plus plug nodes, and the number of gaps, respectively. The last line of the document calls the JavaScript function `E` that will interpret all constituents to expand the document. After this, the document has been fully replaced by the expansion. Note that three script sections are required to ensure that processing occurs in distinct phases and dependencies are resolved correctly. Viewing the HTML source in the browser will display the resulting HTML document, not our encodings.

Our compact representation makes no attempts at actual compression such as `gzip` or XML compression [58], but is highly efficient to encode on the server and to decode in JavaScript on the client. Compression is essentially orthogonal in the sense that our representation works independently of whether or not the transmission protocol compresses documents sent across the network, as shown in Section 14.5. However, the benefit factor of our scheme is of course reduced when compression is added.

14.4.3 Clustering

In `<bigwig>`, the `show` operation is not restricted to transmit a single document. It can be a collection of interconnected documents, called a *cluster*. For instance, a document with input fields can be combined in a cluster with a separate document with help information about the fields.

A hypertext reference to another document in the same cluster may be created using the notation `&x` to refer to the document held in the HTML variable `x` at the time the cluster is shown. When showing a document containing such references, the client can browse through the individual documents without involving the service code. The control-flow in the service code becomes more clear since the interconnections can be set up as if the cluster were a single document and the references were internal links within it.

The following example shows how to set up a cluster of two documents, `input` and `help`, that are cyclically connected with `input` being the main document:

```
service {
  html input = <html>
    Please enter your name: <input name="name"><p>
    Click <a href=[help]>here</a> for help.
  </html>;

  html help = <html>
    You can enter your given name, family name, or nickname.
    <p><a href=[back]>Back</a> to the form.
  </html>;

  html output = <html>Hello <[name]>!</html>;

  session cluster_example() {
    html h, i;
    string s;
    h = help<[back=&i];
    i = input<[help=&h];
    show i receive [s=name];
    show output<[name=s];
  }
}
```

The cluster mechanism gives us a unique opportunity for further reducing network traffic. We can encode the entire cluster as a single JavaScript document, containing all the documents of the cluster along with their interconnections. Wherever there is a document reference in the original cluster, we generate JavaScript code to overwrite the current document in the browser with the referenced document of the cluster. Of course, we also need to add some code to save and restore entered form data when the client leaves and re-enters pages with forms. In this way, everything takes place in the client's browser and the server is not involved until the client leaves the cluster.

14.5 Experiments

Figure 14.7 recounts the experiments we have performed. We have applied our caching technique to the five Web service benchmarks mentioned in the introduction.

In Figure 14.7(b) we show the sizes of the data transmitted to the client. The grey columns show the original document sizes, ranging between 20 and 90 KB. The white columns show the sizes of the total data that is transmitted using our technique, none of which exceeds 20 KB. Of ultimate interest is the black column which shows the asymptotic sizes of the transmitted data, when the templates have been cached by the client. In this case, we see reductions of factors between 4 and 37 compared to the original document size.

The `lycos` benchmark is similar to one presented for HPP [31], except that our reconstruction is of course in `<bigwig>`. It is seen that the size of our residual dynamic data (from 20,183 to 3,344 bytes) is virtually identical to that obtained by HPP (from 18,000 to 3,250 bytes). However, in that solution all caching aspects are hand-coded with the benefit of human insight, while ours is automatically generated by the `<bigwig>` compiler. The other four benchmarks would be more challenging for HPP.

In Figure 14.7(c) we repeat the comparisons from Figure 14.7(b) but under the assumption that the data is transmitted compressed using `gzip`. Of course, this drastically reduces the benefits of our caching technique. However, we still see asymptotic reduction factors between 1.3 and 2.9 suggesting that our approach remains worthwhile even in these circumstances. Clearly, there are documents for which the asymptotic reduction factors will be arbitrarily large, since large constant text fragments count for zero on our side of the scales while `gzip` can only compress them to a certain size. Hence we feel justified in claiming that compression is orthogonal to our approach. When the HTTP protocol supports compression, we represent the string pool in a naive fashion rather than as a trie, since `gzip` does a better job on plain string data. Note that in some cases our uncompressed residual dynamic data is smaller than the compressed version of the original document.

In Figure 14.7(d) and 14.7(e) we quantify the end-to-end latency for our technique. The total download and rendering times for the five services are shown for both the standard documents and our cached versions. The client is Internet Explorer 5 running on an 800 MHz Pentium III Windows PC connected to the server via either a 28.8K modem or a 128K ISDN modem. These are still realistic configurations, since by August 2000 the vast majority of Internet subscribers used dial-up connections [46] and this situation will not change significantly within the next couple of years [64]. The times are averaged over several downloads (plus renderings) with browser caching disabled. As expected, this yields dramatic reduction factors between 2.1 and 9.7 for the 28.8K modem. For the 128K ISDN modem, these factors reduce to 1.4 and 3.9. Even our "worst-case example", `dmodlog`, benefits in this setup. For higher bandwidth dimensions, the results will of course be less impressive.

In Figure 14.7(f) we focus on the pure rendering times which are obtained by averaging several document accesses (plus renderings) following an initial download, caching it on the browser. For the first three benchmarks, our times are in fact a bit faster than for the original HTML documents. Thus, generating a large document is sometimes faster than reading it from the memory cache. For the last two benchmarks, they are somewhat slower. These figures are of course highly dependent on the quality of the JavaScript interpreter that is available in the browser. Compared to the download latencies, the rendering times are negligible. This is why we have not visualized them in Figure 14.7(d) and 14.7(e).

14.6 Future Work

In the following, we describe a few ideas for further cutting down the number of bytes and files transmitted between the server and the client.

In many services, certain templates often occur together in all `show` statements. Such templates could be grouped in the same file for caching, thereby lowering the transmission overhead. In `<bigwig>`, the HTML validation analysis [17] already approximates a graph from which we can readily derive the set of templates that can reach a given `show` statement. These sets could then be analyzed for tightly connected templates using various heuristics. However, there are certain security concerns that need to be taken into consideration. It might not be good idea to indirectly disclose a template in a cache bundle if the show statement does not directly include it.

Finally, it is possible to also introduce language-based server-side caching which is complementary to the client-side caching presented here. The idea is to exploit the structure of `<bigwig>` programs to automatically cache and invalidate the documents being generated. This resembles the server-side caching techniques mentioned in Section 14.2.

14.7 Conclusion

We have presented a technique to revive the existing client-side caching mechanisms in the context of dynamically generated Web pages. With our approach, the programmer need not be aware of caching issues since the decomposition of pages into cachable and dynamic parts is performed automatically by the compiler. The resulting caching policy is guaranteed to be sound, and experiments show that it results in significantly smaller transmissions and reduced latency. Our technique requires no extensions to existing protocols, clients, servers, or proxies. We only exploit that the browser can interpret JavaScript code. These results lend further support to the unique design of dynamic documents in `<bigwig>`.

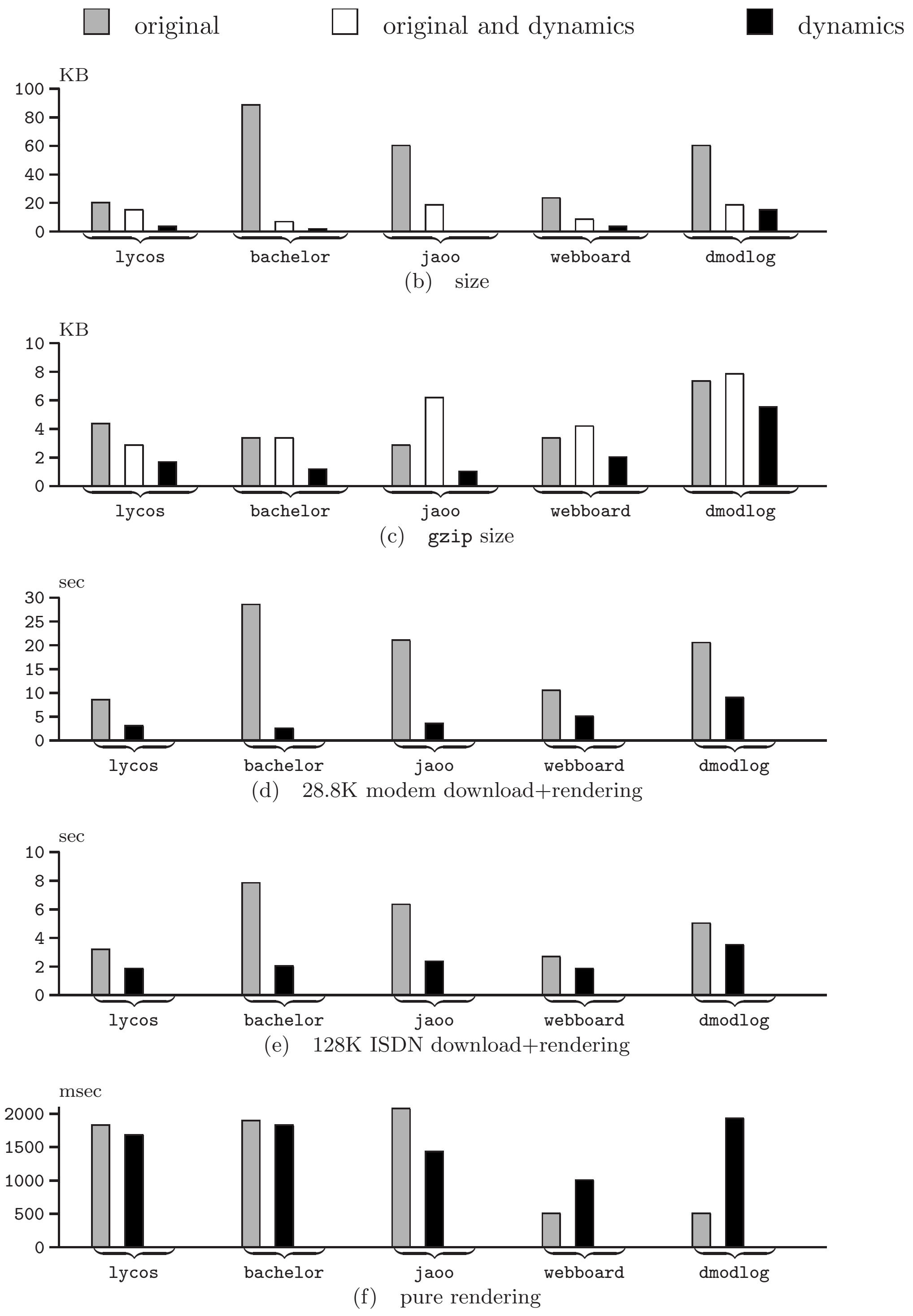


Figure 14.7: Experiments with the template representation.

Chapter 15

Growing Languages with Metamorphic Syntax Macros

with Michael I. Schwartzbach

Abstract

"From now on, a main goal in designing a language should be to plan for growth."
— Guy Steele: Growing a Language, OOPSLA'98 invited talk.

We present our experiences with a syntax macro language which we claim forms a general abstraction mechanism for growing (domain-specific) extensions of programming languages. Our syntax macro language is designed to guarantee *type safety* and *termination*.

A concept of *metamorphisms* allows the arguments of a macro to be inductively defined in a *meta* level grammar and *morphed* into the host language. We also show how the metamorphisms can be made to operate simultaneously on multiple parse trees at once and to accept parse trees as arguments. The result is a highly flexible mechanism for growing new language constructs without resorting to compile-time programming. In fact, whole new languages can be defined at surprisingly low cost.

This work is fully implemented as part of the `<bigwig>` system for defining interactive Web services, but could find use in many other languages.

15.1 Introduction

A compiler with syntax macros accepts collections of grammatical rules that extend the syntax in which a subsequent program may be written. They have long been advocated as a means for extending programming languages [95, 22, 56]. Recent interest in domain-specific and customizable languages poses the challenge of using macros to realize new language concepts and constructs or even to grow entire new languages [79, 10, 59].

Existing macro languages are either unsafe or not expressive enough to live up to this challenge, since the syntax allowed for macro invocations is too restrictive. Also, many macro languages resort to compile-time meta-programming, making them difficult to use safely.

In this paper we propose a new macro language that is at once sufficiently expressive and based entirely on simple declarative concepts like grammars and substitutions. Our contributions are:

- a macro language design with guaranteed *type safety* and *termination* of the macro expansion process;
- a concept of *metamorphism* to allow a user defined grammar for invocation syntax;
- a mechanism for operating simultaneously on *multiple* parse trees;
- a full and *efficient implementation* for a syntactically rich host language; and
- a *survey* of related work, identifying and classifying relevant properties;

This work is carried out in the context of the `<bigwig>` project [74], but could find uses in many other host languages for which a top-down parser can be constructed. For a given application of our approach, knowledge of the host grammer is required. However, no special properties of such a grammar are used. In fact, it is possible to build a generator that for a given host grammar automatically will provide a parser that supports our notion of syntax macros.

15.2 Related Work Survey

Figure 15.1 contains a detailed survey of the predominant macro languages that have previously been proposed. We have closely investigated the following eight macro languages and their individual semantic characteristics: the C preprocessor, `CPP` [49, 78]; the Unix macro preprocessor, `M4`; TEX's built-in macro mechanism; the macro mechanism of `Dylan` [76]; the `C++ templates` [80]; `Scheme`'s hygienic macros [48, 53]; the macro mechanism of the Jakarta Tool Suite, `JTS` [10]; and the Meta Syntactic Macro System, `MS`^{2} [95]. The JSE system [6] is a version of `Dylan` macros adapted to Java and is not treated independently here. This survey has led us to identify and group 32 properties that characterize a macro language and which we think are relevant for comparing such work. Our own macro language is designed by explicitly considering exactly those properties; for comparison, it is included in the last column of the survey table.

	Property \ Language	`CPP`	`M4`	TEX	`Dylan`	`C++ templates`	`Scheme`	`JTS`	`MS`^{2}	`<bigwig>`
Gen.	Level of operation	lexical	lexical	lexical	hybrid	syntactical	syntactical	syntactical	syntactical	**syntactical**
	Language dependent	no	no	yes	yes	yes	yes	yes	yes	**yes**
	Programmable	conditionals	arithmetic	yes	no	constant folding	yes	no	yes	**no**
Syntax	Definition keyword	`#define`	`define`	`\def`	`define macro`	`template`	`define-syntax`	`macro`	`syntax`	`macro`
	Formal argument def	*id*	N/A	`#1` to `#9`	*?id:id*, *?:id*, *?id*	`<`*nt id*`>`	*id*	*nt id*	`$$`*nt*`::`*id*, `$$`...`::`*id*	`<`*nt id*`>`, `<`*id*`:` *nt id*`>`
	Formal argument use	*id*	`$0` to `$9`	`#1` to `#9`	*?id*	*id*	*id*	*id*	`$`*id*	`<`*id*`>`
	Invocation syntax	*id*`(_, _, _)`	*id*`(_, _, _)`	`\`*id* ...	*id* ...	*id*`<_, _, _>`	`(`*id* `_ _ _)`	`#`*id*`(_, _, _)`	*id* ...	*id* **...**
Type	Argument types declared	N/A	N/A	N/A	yes	yes	implicitly	yes	yes	**yes**
	Argument nonterminals	N/A	N/A	N/A	7+*token*	*id, type, const*	*s-exp*	6	15	**all 55**
	Argument types checked	N/A	N/A	N/A	yes	yes	yes	yes	yes	**yes**
	Result types declared	N/A	N/A	N/A	yes	no	implicitly	yes	yes	**yes**
	Result nonterminals	N/A	N/A	N/A	*stm, fcall, def*	*decl*	*s-exp*	5	15	**all 55**
	Result types checked	N/A	N/A	N/A	no	N/A	no	yes	yes	**yes**
Definition	Multiple definitions	no	no	no	yes	yes	yes	no	no	**yes**
	Definition selection	N/A	N/A	N/A	order listed	specificity	order listed	N/A	N/A	**specificity**
	Definition scope	one pass	one pass	one pass	one pass	one pass	one pass	one pass	two pass	**two pass**
	Undefine	yes	redefine	redefine	no	no	redefine	no	N/A	**N/A**
	Local macro definitions	no	yes	yes	no	yes	yes	yes	no	**yes**
	Direct recursion	no	yes	yes	yes	no	yes	no	no	**rejected**
	Indirect recursion	no	yes	yes	yes	yes	yes	N/A	yes	**N/A**
	Argument structure	fixed	fixed	fixed	grammar	fixed	list	fixed	option, list, tuple	**grammar**
Invocation	Body expansion	lazy	eager	lazy	lazy	lazy	lazy	eager	eager	**eager**
	Order of expansion	prescan	prescan	outer	prescan	N/A	outer	inner	outer	**inner**
	Parsing ambiguities	N/A	N/A	shortest	shortest	N/A	N/A	N/A	greedy	**greedy**
	Hygienic expansion	no	no	no	yes	no	yes	(yes)	no	**yes**
	Macros as results	no	yes	yes	no	no	yes	yes	yes	**no**
	Guaranteed termination	yes	no	no	no	no	no	yes	no	**yes**
Impl.	Transparent	yes	N/A	yes	yes	yes	yes	yes	yes	**yes**
	Error trailing	N/A	N/A	no	no	no	yes	no	no	**yes**
	Pretty printing	no	no	no	no	no	yes	no	no	**yes**
	Package Mechanism	no	no	no	no	no	yes	no	no	**yes**

Figure 15.1: A macro language survey.

15.2.1 General Properties

The paramount characteristic of a macro language is whether it operates at the *lexical* or *syntactical* level. Lexical macro languages allow tokens to be substituted by arbitrary sequences of characters or tokens. These definitions may be parameterized so that the substitution sequence contains placeholders for the actual parameters that are themselves just arbitrary character sequences. `CPP`, `M4`, and TEX are well-known lexical macro languages. Conceptually, lexical macro processing precedes parsing and is thus ignorant of the syntax of the underlying host language. In fact, `CPP` and `M4` are language independent preprocessors for which there is no concept of host language. As a direct consequence of syntactic independence, all lexical macro languages share many dangers that can only be avoided by clever hacks and workarounds, which are by now folklore.

A representative example is the following `square` macro:

```
#define square(X) X*X
```

which works as expected in most cases. However, if invoked with the argument `z+1` the result will be the character sequence `z+1*z+1` which is interpreted as `z+(1*z)+1`. A solution to this particular problem is explicitly to add parentheses around the arguments to control subsequent parsing:

```
#define square(X) (X)*(X)
```

A more subtle problem arises when invoking the following macro:

```
#define swap(X,Y) { int t=X; X=Y; Y=t; }
```

in this context:

```
if (a>b) swap(a,b); else a = 0;
```

This program gives an unexpected "`parse error before 'else'`" because there are two statements between the keywords `if` and `else`. The first is the compound statement from the expansion of the `swap` macro, the second is the empty statement (the semicolon) following the invocation of the `swap` macro. A workaround employed by skilled `CPP` programmers is to rewrite the macro body as a `do-while` construct without a terminating semicolon and with a constant false condition:

```
#define swap(X,Y) do { int t=X; X=Y; Y=t; } while (0)
```

Now, invocations of the `swap` macro can be safely terminated with a semicolon, expanding to one statement only. These are the kind of low-level issues that plague lexical macro programmers.

In contrast, syntactical languages operate on parse trees, as depicted in Figure 15.2, which of course requires knowledge of the host language and its grammar. Syntactical macro languages include `C++ templates`, `Scheme`, `JTS`,

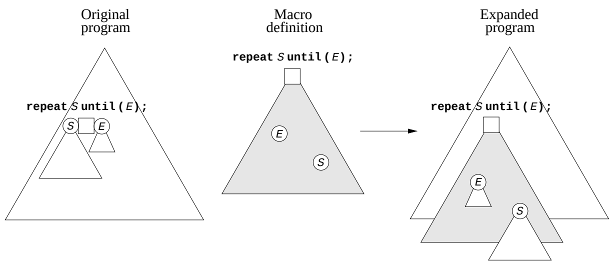


Figure 15.2: Syntax macros—operators on parse trees.

and `MS`^{2}. The language `Dylan` is a hybrid that operates simultaneously on token streams and parse trees.

Some macro languages allow explicit *programming* on the parse trees that are being constructed, while others only use pattern matching and substitution. `CPP` only allows simple conditionals, `M4` offers simple arithmetic, `C++ templates` performs constant folding (which together with multiple definitions provide a Turing-complete compile-time programming language [90]), while `Scheme` and `MS`^{2} allow arbitrary computations.

15.2.2 Syntax Properties

The syntax for defining and invoking macros varies greatly. The main point of interest is how liberal an invocation syntax is allowed. At one end of the spectrum is `CPP` which requires parenthesized and comma separated actual arguments, while at the other end `Dylan` allows an almost arbitrary invocation syntax following an initial identifier.

15.2.3 Type Properties

There are two notions of *type* in conjunction with syntactical macro languages, namely *result types* and *argument types*, both ranging over the *nonterminals* of the host language grammar. These are often explicitly declared, by naming nonterminals of some standardized host language grammar. Using these, syntactical macro languages have the possibility of type checking definitions and invocations. Definitions may be checked to comply with the declared nonterminal return type of the macro, assuming that the placeholders have the types dictated by the arguments. Invocations may be checked to ensure that all arguments comply with their declared types. Often the argument type information is used to guide parsing, in which case this last check comes for free. If both checks are performed, no parse errors can occur as a direct consequence of macro expansion.

Only `JTS` and `MS`^{2} take full advantage of this possibility. The others mentioned fall short in various ways, for example by not checking that the macro body conforms to the result nonterminal. The languages also differ in how many nonterminals from the host grammar can be used as such types.

15.2.4 Definition Properties

There are many relevant properties of macro definitions. The languages `Dylan`, `CPP`, and `Scheme`, allow more than one macro to be defined with the same name; a given invocation then selects the appropriate definition either by trying them out in the order listed or by using a notion of *specificity*.

Most macro languages have *one-pass* scope rules for macro definitions, meaning that a macro is visible from its lexical point of definition and onward. Only `MS`^{2} employs a *two-pass* strategy, in which macro definitions are available even before their lexical point of definition. With one-pass scope rules, the order in which macros are defined is significant, whereas with two-pass scope rules the macro definitions may be viewed as a *set*. The latter has the nice property that the definition order can be rearranged without affecting the semantics. However, this is not completely true of `MS`^{2} since its integrated compile-time programming language has one-pass scope rules. Some of the languages allow macros to be undefined or redefined which of course only makes sense in the presence of one pass scope rules. Many languages permit local macro definitions, but `CPP`, `Dylan`, and `JTS` have no such concept.

There are two kinds of macro recursion; *direct* and *indirect*. Direct recursion occurs when the body of a macro definition contains an invocation of itself. This always causes non-termination. Indirect recursion occurs when a self-invocation is *created* during the expansion. This can either be the result of a compile-time language *creating* a self-invocation or the result of the expansion being reparsed as in the *prescan* expansion strategy (see below). Without a compile-time programming language with side-effects to "break the recursion", indirect recursion also causes non-termination. The above generalizes straightforwardly to mutual recursion. Most of the languages tolerate some form of macro recursion, only `CPP` and `JTS` completely and explicitly avoid recursion.

An important issue is the argument structure that is allowed. Most languages require a fixed number of arguments for each macro. `Scheme` allows lists of arguments, `MS`^{2} allows lists, tuples, and optional arguments, while `Dylan` is the most flexible by allowing the argument syntax to be described by a user defined grammar.

15.2.5 Invocation Properties

A macro body may contain further macro invocations. The languages are evenly split as to whether a macro body is expanded *eagerly* at its definition or *lazily* at each invocation. An eager strategy will find all errors in the macro body at definition time, even if the macro is never invoked.

Similarly, the actual arguments may contain macro invocations; here, the languages split on using an *inner* or *outer* expansion strategy. However, `CPP`, `M4`, and `Dylan` use a more complex strategy known as *argument prescan*. When a macro invocation is discovered, all arguments are parsed and any macros inside are invoked. These expanded arguments are then substituted for their placeholders in a copy of the macro body. Finally, the entire result is *rescanned*, processing any newly produced macro invocations. Note that this strategy only makes sense for lexical macro languages.

The languages that allow a liberal invocation syntax where the arguments are not properly delimitered sometimes face ambiguities in deciding how to match actual to formal macro arguments. The lexical languages, TEX and `Dylan`, resolve such ambiguities by chosing the *shortest* possible match; in contrast, the syntactical language `MS`^{2} employs a *greedy* strategy that for each formal argument parses as much as possible. None of the languages investigated employed *back-tracking* for matching invocations with definitions.

Most syntactical languages use automatic α-conversion to obtain *hygienic* macros; `MS`^{2} requires explicit renamings to be performed by the programmer. Several languages allow new macro definitions to be generated by macro expansions. Only `CPP` and `JTS` guarantee termination of macro expansion; the others fail either by a naive treatment of recursive macros or by allowing arbitrary computations during expansion.

15.2.6 Implementation Properties

Macro languages are generally designed to be *transparent*, meaning that subsequent phases of the compilation need not be aware of macro expansions. However, none apart from `Scheme` seem to allow *pretty printing* of the unexpanded syntax and *error trailing*, meaning that errors from subsequent phases are traced back to the unexpanded syntax. Finally, a *package* concept for macros seems again only to be considered by Scheme [92].

15.2.7 Other Related Work

Our macro language shares some features of a previous work on extensible syntax [24], although that is not a macro language. Rather, it is a framework for defining new syntax that is represented as parse tree data structures in a *target* language, in which type checking and code generation is then performed. In contrast, our new syntax is directly translated into parse trees in a *host* language. Also, the host language syntax is always available on equal footing with the new syntax. However, the expressiveness of the extensible syntax that is permitted in [24] is very close to the argument syntax that we allow, although there are many technical differences, including definition selection, parsing ambiguities, expansion strategy, and error trailing. Also, we allow a more general translation scheme.

15.3 Designing a Macro Language

The ideal macro language would allow *all* nonterminals of the host language grammar to be extended with arbitrary new productions, defining new constructs that appear to the programmer as if they were part of the original language. The macro languages we have seen in the previous section all approximate this, some better than others.

In this section we aim to come as close to this ideal as practically possible. Later, we take a further step by allowing the programmer to define also new nonterminals. Another goal is to obtain a *safe* macro language, where type checking and termination are guaranteed. We will carefully consider the semantic aspects identified in Figure 15.1 in our design.

15.3.1 Syntax

Our syntax macro language looks as follows:

macro	:	`macro` `<`*nonterm*`>` *id* $\langle$*param*$\rangle^*$ `::=` `{` *body* `}`
param	:	*token*
	|	`<`*nonterm id*`>`

A syntax macro has four constituents: a *result type* (which is a nonterminal of the host grammar), an identifier *naming* the macro, a *parameter list* specifying the invocation syntax, and a *body* that must comply with the result type.

The result type declares the type of the body and thereby the syntactic contexts in which invocations of the macro are permitted. Adhering to Tennent's *Principle of Abstraction* [83], we allow *nonterm* to range over *all* nonterminals of the host language grammar. Of course, the nonterminals are from a particular standardized abstract grammar. In the case of the `<bigwig>` host language, 55 nonterminals are available.

As in MS^2, a macro must start with an identifier. It is technically possible to lift this restriction [70], but it serves to make macro invocations easier to recognize. The parameter list determines the rest of the invocation syntax. Here, we allow arbitrary tokens interspersed among arguments that are identifiers typed with nonterminals. The list ends with the "`::=`" token. The macro body enclosed in braces conforms to the result type and references the arguments through identifiers in angled brackets.

Simple Examples

A simplest possible macro without arguments is:

```
macro <floatconst> pi ::= {
  3.1415927
}
```

whose invocation `pi` is only allowed in places where a *floatconst* may appear. The next macro takes an argument and executes it with 50% probability:

```
macro <stm> maybe <stm S> ::= {
  if (random(2)==1) <S>
}
```

A more interesting invocation syntax is:

```
macro <stm> repeat <stm S> until (<exp E>); ::= {
  {
    bool first = true;
    while (first || !<E>) {
      <S>
      first = false;
    }
  }
}
```

which extends the host language with a `repeat` construct that looks and feels exactly like the real thing. Identifiers such as `repeat` and `until` are even treated as keywords in the scope of the macro definition. The semantic correctness of course relies on α-conversion of `first`. Incidentally, this is the macro used in Figure 15.2.

An example with multiple definitions supplies a Francophile syntax for existing constructs:

```
macro <stm> si (<exp E>) <stm S> ::= {
  if (<E>) <S>
}

macro <stm> si (<exp E>) <stm S> sinon <stm S2> ::= {
  if (<E>) <S> else <S2>
}
```

The two definitions are both named `si` but have different parameters.

Macro Packages

Using macros to enrich the host language can potentially create a Babylonic confusion. To avoid this problem, we have created a simple mechanism for scoping and packaging macro definitions. A package containing macro definitions is viewed as a *set*, that is, we use two pass scope rules where all definitions are visible to each other and the order is insignificant. A dependency analysis intercepts and *rejects* recursive definitions.

A package may `require` or `extend` other packages. Consider a package P that contains a set of macro definitions M, requires a package R, and extends another package E. The definitions visible inside the bodies of macros in M are $M \cup R \cup E$ and those that are exported from P are $M \cup E$. Thus, `require` is used for obtaining *local* macros. The strict view that a package defines a *set* eliminates many potential problems and confusions.

15.3.2 Parsing Definitions

Macro definitions are parsed in two passes yielding a set of definitions. First, the macro headers are collected into a structure that will later guide the parsing of invocations. The bodies are lexed to discover macro invocations from which a dependency graph is constructed. Second, the macro bodies are parsed in topological order to respect these dependencies. To ensure termination, we intercept and reject cycles. The result is for each body a parse tree that conforms to the result type and contains placeholder nodes for occurrences of arguments. It is checked that the body can be derived from the result nonterminal when the placeholders are assumed to be derived from the corresponding argument nonterminals. Note that this yields an eager expansion strategy allowing parse errors in the macro body to be reported at definition time.

15.3.3 Parsing Invocations

Macro invocations are detected by the occurrence of an identifier naming a macro. At this point, the parser determines if the result type of the macro is reachable from the current point in parsing. If not, parsing is aborted. Otherwise, parsing is guided to this nonterminal and *invocation parsing* begins. The result is a parse tree that is inserted in place of the invocation.

Invocation parsing is conducted by interpreting the macro parameter list, matching required tokens and collecting actual argument parse trees. When the end of the parameter list is reached, the actual arguments are substituted into the placeholders in a copy of the macro body. This process is commonly referred to as *macro expansion.* The parsing is *greedy* since an actual argument is parsed as far as possible in the usual top-down parsing style.

However, this basic mechanism is not powerful enough to handle *multiple* definitions of a macro which yields a more flexible invocation syntax and are crucial for the metamorphisms presented later. For that purpose, we must interpret a *set* of parameter lists. We base the definition selection on the concept of *specificity* which is independent of the macro definition order. This is done by gradually challenging each parameter list with the input tokens. There are three cases for a challenge:

- if a list is empty, then it always survives;
- if a list starts with a token, then it survives if it equals the input token; and
- if a list starts with an argument `<`N a`>`, then it survives if the input token belongs to first(N) in the host grammar.

Several parameter lists may survive the challenge. Among those, we only keep the most *specific* ones. The empty list is always eliminated unless all lists are empty. Among a set of non-empty lists, the survivors are those whose first parameter is maximal in the ordering $p \sqsubset q$ defined as $\phi(q) \subset \phi(p)$, where $\phi(\textit{token})$ is the singleton $\{\textit{token}\}$ and $\phi($`<`$N\ a$`>`$)$ is first(N) in the host grammar. The tails of the surviving lists are then challenged with the next input token, and so on.

The intuition behind our notion of specificity can be summarized in a few rules of thumb: 1) always prefer longer parameter lists to shorter ones, 2) always prefer a token to a nonterminal, 3) always prefer a narrow nonterminal to a wider one. Rule 1) is the reason that the dangling `sinon` problem for our Francophile example is solved correctly. This strategy has a far reaching generality that also works for the metamorph rules introduced in Section 15.5.

The following example illustrates how invocations are parsed. Consider the macro definitions:

```
macro <exp> sync [ 0 : <id I> ] ::= { ... }

macro <exp> sync [ 0 : <exp E> ] ::= { ... }

macro <exp> sync [ <exp E> ] ::= { ... }

macro <exp> sync ::= { ... }
```

The invokation that we must parse is: `sync[0:x]`. In the first challenge round we have the situation:

sync	[	0	:	<id I>	]
sync	[	0	:	<exp E>	]
sync	[	<exp E>	]		
sync					
sync	[	0	:	x	]
↑					

All macro headers survive, since they all match the `sync` token. In the next challenge round:

sync	[	0	:	<id I>	]
sync	[	0	:	<exp E>	]
sync	[	<exp E>	]		
~~sync~~					
sync	[	0	:	x	]
	↑				

the shortest macro header loses, since the others are prepared to carry on with the `[` token. In the next round:

sync	[	0	:	<id I>	]
sync	[	0	:	<exp E>	]
~~sync~~	~~[~~	~~<exp E>~~	~~]~~		
~~sync~~					
sync	[	0	:	x	]
		↑			

only the first two macro headers survive, since they match the token `0` with tokens rather than the more general `exp` non-terminal. In the fourth challenge round they agree and both survive:

sync	[	0	:	<id I>	]
sync	[	0	:	<exp E>	]
~~sync~~	~~[~~	~~<exp E>~~	~~]~~		
~~sync~~					
sync	[	0	:	x	]
			↑		

In the fifth challenge round:

`sync`	`[`	`0`	`:`	`<id I>`	`]`
~~`sync`~~	~~`[`~~	~~`0`~~	~~`:`~~	~~`<exp E>`~~	~~`]`~~
~~`sync`~~	~~`[`~~	~~`<exp E>`~~	~~`]`~~		
~~`sync`~~					

`sync`	`[`	`0`	`:`	`x`	`]`
				↑	

the first macro header is declared the winner, since it matches the token `x` with the non-terminal `id` and `id` $\sqsubset$ `exp` (since first(`id`) $\subset$ first(`exp`)). This chosen macro header survives through the remaining `]` token and its expansion is then performed.

For the *order of expansion* we have chosen the *inner* strategy. Since our macros are terminating, the expansion order is semantically transparent, apart from a subtle difference with respect to α-conversion. The inner strategy is more efficient since arguments are only parsed once.

15.3.4 Well-Formedness

A set of macros with the same name must be *well-formed.* This means that they must all have the same result type. Actually, this restriction could be relaxed to allow different return types for macros with the same name by using a contravariant specificity ordering to determine which one to invoke. Furthermore, to guarantee that the challenge rounds described above have a unique final winner, we impose the requirement that for all pairs of parameter lists of the form $\pi p_1 \pi_1$ and $\pi p_2 \pi_2$, then

$$\phi(p_1) \setminus \phi(p_2) = \emptyset \ \vee \ \phi(p_2) \setminus \phi(p_1) = \emptyset \ \vee \ \phi(p_1) \cap \phi(p_2) = \emptyset$$

and if $\phi(p_1)$ equals $\phi(p_2)$ then p_1 must equal p_2.

15.3.5 Hygienic Macros

To achieve hygienic macros, we automatically α-convert all identifiers inside macro bodies during expansion. Unlike `Scheme` [52, 28, 33], we also α-convert *free* identifiers, since they cannot be guaranteed to bind to anything sensible in the context of an invocation. As we thus α-convert *all* identifiers, the macro needs only recognize all parse tree nodes of nonterminal *id*; that is, no symbol table information is required. To communicate identifiers from the invocation context we encourage the macro programmer to supply those explicitly as arguments of type *id*. If an unsafe free variable is required, it must be *backpinged* to avoid α-conversion. It is often necessary to use computed identifiers, as seen in Figure 15.6. For that purpose, we introduce an injective and associative binary concatenation operator "~" on identifiers. The inductive predicate α determines if an identifier will be α-converted:

- $\alpha($‘$i) = \mathit{false}$;
- $\alpha(i\,\tilde{}\,j) = \alpha(i) \wedge \alpha(j)$;
- $\alpha($`<`i`>`$) = \mathit{false}$, if `<`i`>` is an argument of type *id*; and

- $\alpha(i) = \mathit{true}$, otherwise.

The following example illustrates the effect of α-conversion during macro expansion:

```
                              first = 0;
                              { bool first~1 = true;
                                while (first~1 || !(first > 7)) {
                                  { first++;
first = 0;                          { bool first~2 = true;
repeat {                              while (first~2 || !(i > 10)) {
  first++;                              { i++;
  repeat {                                f(i);
    i++;                    ⇒           }
    f(i);                               first~2 = false;
  } until (i>10);                     }
} until (first>7);                  }
                                  }
                                  first~1 = false;
                                }
                              }
```

15.4 Growing Language Concepts

Our macro language allows the host language to grow, not simply with handy abbreviations but with new concepts and constructs. Our host language, `<bigwig>`, is designed for programming interactive Web services and has a very general mechanism for providing concurrency control between session threads [71, 13]. The programmer may declare labels in the code and use temporal logic to define the set of legal traces for the entire service. This is a bit harsh on the average programmer and consequently a good opportunity for using macros.

Figure 15.6 shows a whole stack of increasingly high-level concepts that are introduced on top of each other, profiting from the possibility to define macros for all nonterminals of the host language. Details of the `<bigwig>` syntax need not be understood. The `allow`, `forbid`, and `mutex` macros abbreviate common constructs in temporal logic and produce results of type *formula*. The macro `region` of type *toplevel* is different; it introduces a new concept of *regions* that are declared on equal footing with other native concepts. The `exclusive` macro of type *stm* defines a new control structure that secures exclusive access to a previously declared region. The `resource` macro of type *toplevel_list* declares an instance of another novel concept that together with the macros `reader` and `writer` realizes the *reader/writer protocol* for specified resources. Finally, the `protected` macro seemingly provides a *modifier* that allows any declared variable to be subject to that protocol. The macros all build on top of each other and produce no less than six levels of abstraction as depicted in Figure 15.3. A similar development could have implemented other primitives, such as semaphores, monitors, and fifo pipes. This demonstrates how the host language becomes highly tailorable with very simple means. The `<bigwig>` language employs an extensive collection of predefined macros to enrich the core language.

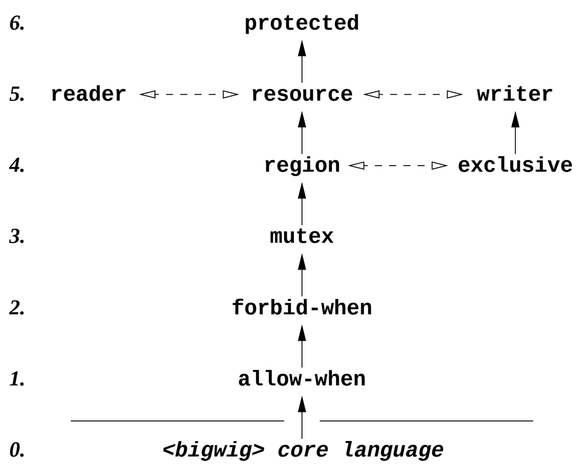


Figure 15.3: A stack of macro abstractions.

An example of a program using the high-level abstractions is:

```
service {
  protected shared int counter;

  html Doc = <html>
    You are visitor number <b><[number]></b>
  </html>;

  session Access() {
    html D;
    reader (counter) D = Doc <[number=counter];
    writer (counter) counter++;
    exit D;
  }
}
```

This program is a Web service that shows a page with the ubiquitous page counter which is declared using the `protected` macro. When a client issues a request to run the session `Access`, the value of the counter is read inside a `reader` region and a document showing this value is assembled. Subsequently, the counter is incremented in a `writer` region. Finally, the document is transmitted to the client.

15.5 Metamorphisms

Macro definitions specify two important aspects: the *syntax definitions* characterizing the syntactic structure of invocations and the *syntax transformations* specifying how "new syntax" is *morphed* into host language syntax.

So far, our macros can only have a finite invocation syntax, taking a fixed number of arguments each of which is described by a host grammar nonterminal. In the following we will move beyond this limitation, focusing initially on the syntax definition aspects.

The previously presented notion of *multiple* definitions allow macros with varying arity. The following example defines an `enum` macro as known from C that takes one, two, or three identifier arguments:

```
macro <decls> enum { <id X> } ; ::= {
  const int <X> = 0;
}

macro <decls> enum { <id X> , <id Y> } ; ::= {
  const int <X> = 0;
  const int <Y> = 1;
}

macro <decls> enum { <id X> , <id Y> , <id Z> } ; ::= {
  const int <X> = 0;
  const int <Y> = 1;
  const int <Z> = 2;
}
```

Evidently, it is not possible to define macros with arbitrary arity and the specifications exhibit a high degree of redundancy. In terms of syntax definition, the three `enum` definitions correspond to adding three *unrelated* right-hand side productions for the nonterminal *decls*:

$$\begin{array}{lcl} \textit{decls} & : & \texttt{enum \{ } \textit{id} \texttt{ \} ;} \\ & | & \texttt{enum \{ } \textit{id} \texttt{ , } \textit{id} \texttt{ \} ;} \\ & | & \texttt{enum \{ } \textit{id} \texttt{ , } \textit{id} \texttt{ , } \textit{id} \texttt{ \} ;} \end{array}$$

`Scheme` amends this by introducing a special ellipsis construction, "`...`" to specify lists of nonterminal s-expressions. `MS`^{2} moves one step further by permitting also tuples and optional arguments, corresponding to allowing the use of regular expressions over the terminals and nonterminals of the host grammar on the right-hand sides of productions. The ubiquitous EBNF syntax is available for designating options "`?`", lists "`*`" or "`+`", and tuples "`{...}`" (for grouping). In addition, `MS`^{2} provides a convenient variation of the Kleene star for specifying token-separated lists of nonterminals. Here, we use $N^{\oplus}$ as notation for one-or-more *comma separated* repetitions of the nonterminal N. An `enum` macro defined via this latter construction corresponds to extending the grammar as follows:

$$\textit{decls} \quad : \quad \texttt{enum \{ } \textit{id}^{\oplus} \texttt{ \} ;}$$

The `Dylan` language has taken the full step by allowing the programmer to describe the macro invocation syntactic structure via a user defined *grammar*, permitting the introdution of new user defined nonterminals. This context-free language approach is clearly more general than the regular language approach, since it can handle balanced tree structures. The `enum` invocation syntax could be described by the following grammar fragment that introduces a user defined nonterminal called $\underline{\textit{enums}}$ (underlined for readability):

$$\begin{array}{lcl} \textit{decls} & : & \texttt{enum \{}\ \textit{id}\ \underline{\textit{enums}}\ \texttt{\} ;} \\ \underline{\textit{enums}} & : & \texttt{,}\ \textit{id}\ \underline{\textit{enums}} \\ & | & \varepsilon \end{array}$$

In Dylan, the result of parsing a user defined nonterminal also yields a result that can be substituted into the macro body. However, this result is an *unparsed* chunk of tokens with all the associated lexical macro language pitfalls.

We want to combine this great definition flexibility with type safety. Thus, we need some way of specifying and checking the *type* of the result of parsing a user defined nonterminal. Clearly, such nonterminals cannot exist on an equal footing with those of the host language; a syntax macro must ultimately produce host syntax and thus cannot return user defined ASTs. To this end, we associate to every user defined nonterminal a host nonterminal result type from which the resulting parse tree must be derived. Thus, the syntax defined by the user defined nonterminals is always morphed directly into host syntax. The specification of this morphing is inductively given for each production of the grammar. In contrast, $\texttt{MS}^2$ relies on programming and computation for specifying and transforming their regular expressions of nonterminals into parse trees.

To distinguish clearly from the host grammar, we call the user defined nonterminal productions typed with host nonterminals for *metamorphisms*. A metamorphism is a rule specifying how the macro syntax is *morphed* into host language syntax. The syntax for macro definitions is generalized as follows to accommodate the metamorphisms:

$$\begin{array}{lcl} \textit{macro} & : & \texttt{macro <}\textit{nonterm}\texttt{>}\ \textit{id}\ \langle\textit{param}\rangle^*\ \texttt{::= \{}\ \textit{body}\ \texttt{\}} \\ & | & \texttt{metamorph <}\textit{nonterm}\texttt{>}\ \underline{\textit{id}}\ \texttt{-->}\ \langle\textit{param}\rangle^*\ \texttt{::= \{}\ \textit{body}\ \texttt{\}} \\ \textit{param} & : & \textit{token} \\ & | & \texttt{<}\textit{nonterm id}\texttt{>} \\ & | & \texttt{<}\underline{\textit{id}}\texttt{:}\ \textit{nonterm id}\texttt{>} \end{array}$$

We have introduced two new constructs. A parameter may now also be of the form <$\underline{M}$: N a>, meaning that it is named a, has an invocation syntax that is described by the metamorph nonterminal $\underline{M}$, and that its result has type N. The metamorph syntax and the inductive translation into the host language is described by the metamorph rules. To the left of the "-->" token is the result type and name of the metamorph nonterminal, and to the right is a parameter list defining the invocation syntax and a body defining the translation into the host language. The metamorph rules may define an arbitrary grammar. In its full generality, a metamorph rule may produce multiple results each defined by a separate body.

We are now ready to define the general enum macro in our macro language. The three production rules above translates into the following three definitions:

```
macro <decls> enum { <id I> <enums: decls Ds> } ; ::= {
  int e = 0;
  const int <I> = e++;
  <Ds>
}
```

```
metamorph <decls> enums --> , <id I> <enums: decls Ds> ::= {
  const int <I> = e++;
  <Ds>
}

metamorph <decls> enums --> ::= {}
```

The first rule defines a macro `enum` with the metamorph argument `<`*enums*`:` *decls* `Ds>` describing a piece of invocation syntax that is generated by the nonterminal *enums* in the metamorph grammar. However, *enums* parse trees are never materialized, since they are instantly morphed into parse trees of the nonterminal *decls* in the host grammar.

The body of our `enum` macro commences with the declaration of a variable `e` used for enumerating all the declared variables at runtime. This declaration is followed by the morphing of the (first) identifier `<I>` into a constant integer declaration with initialization expression `e++`. Then comes `<Ds>` which is the *decls* result of metamorphing the remaining identifiers to constant integer declarations.

The next two productions in the `enum` grammar translates into two metamorph definitions. The first will take a comma and an identifier followed by a metamorph argument and morph the identifier into a constant integer declaration as above and return this along with whatever is matched by another metamorph invocation. The second metamorph definition offers a termination condition by parsing nothing and returning the empty declarations.

For simplicity, the constant integer declarations in the bodies of the first two rules are identical. This redundance can be alleviated either by placing this constant declaration in the body of another macro or by introducing another metamorphism returning the declaration at the place of the identifiers.

The next example shows how the invocation syntax of a `switch` statement syntax is easily captured and desugared into nested `if` statements:

```
macro <stm> switch (<exp E>) { <swbody: stm S> } ::= {
  {
    typeof(<E>) x = <E>;
    <S>
  }
}

metamorph <stm> swbody -->
    case <exp E>: <stms Ss> break; <swbody: stm S> ::= {
  if (x==<E>) { <Ss> } else <S>
}

metamorph <stm> swbody --> case <exp E>: <stms Ss> break; ::= {
  if (x==<E>) { <Ss> }
}
```

15.5.1 Parsing Invocations

The strategy for parsing invocations is unchanged. The $\sqsubset$ order is generalized appropriately by defining $\phi($`<`$\underline{M}$`:` $N\ a$`>`$)$ to be first($\underline{M}$) in the metamorph grammar. Note that it is always possible to abbreviate part of the invocation syntax by introducing a new metamorph nonterminal while preserving the semantics.

15.5.2 Well-Formedness

As for syntax macros, the set of productions for a given metamorph nonterminal must be well-formed. Furthermore, to ensure termination of our greedy strategy, we prohibit left-recursion in the metamorph grammar. Finally, we include the sanity check that each metamorph nonterminal must derive some finite string.

15.5.3 Hygienic Macros

Metamorph productions do not initiate α-conversion. This is only done on the entire body of a syntax macro, conceptually *after* its metamorphic arguments have been substituted. This is seen in the `enum` example, where the expansion of "`enum {d,e};`" is:

```
int e~42 = 0;
const int d = e~42++;
const int e = e~42++;
```

In this resulting parse tree, the local occurrence of `e` is everywhere α-converted to the same `e~42`, which is necessary to yield the proper semantics.

15.6 Multiple Results

In its full generality, a metamorph production may morph the invocation syntax into *several* resulting parse trees in the host grammar. This can be seen as a generalization of the `divert` primitive from `M4`; however, our solution statically guarantees type safety of the combined result. The `metamorph` rules and metamorph formals are extended to cope with multiple returns and arguments:

macro : `metamorph <`$\langle$*nonterm*$\rangle^{\oplus}$`>` $\underline{id}$ `-->` $\langle$*param*$\rangle^{*}$ `::=` $\langle$`{` *body* `}`$\rangle^{+}$
param : `<`$\underline{id}$`:` $\langle$*nonterm id*$\rangle^{\oplus}$`>`

The following example illustrates in a simple way how multiple metamorph results add expressive power to our macro language. We define a macro `reserve` that takes a variable number of identifiers denoting resources and a statement. The macro abstraction will acquire the resources in the order listed, execute the statement, and release the resources in reverse order.

```
macro <stm> reserve ( <id X> <res: stms Ss1, stms Ss2> ) <stm S> ::= {
  { acquire(<X>); <Ss1> <S> <Ss2> release(<X>); }
}

metamorph <stms,stms> res --> , <id X> <res: stms Ss1, stms Ss2> ::= {
  acquire(<X>); <Ss1>
}{
  <Ss2> release(<X>);
}

metamorph <stms,stms> res --> ::= {}{}
```

With these definitions, the macro expands as follows:

```
                                     acquire(db);
                                     acquire(master);
reserve (db, master, slave) {        acquire(slave);
  ...                           ⇒    ...
}                                    release(slave);
                                     release(master);
                                     release(db);
```

Without multiple results, some transformations are impossible or require contorted encodings.

15.7 Metamorph Arguments

It is possible to add typed arguments to metamorphisms while retaining safety. This permits context-sensitive transformations in the sense that parse trees may be constructed and supplied to inner metamorph invocations.

To this end, we extend the syntax for metamorph definitions as follows:

```
macro   : metamorph <⟨nonterm⟩^⊕> id formals^? --> ⟨param⟩^* ::= ⟨{ body }⟩^+
param   : <id: ⟨nonterm id⟩^⊕> actuals^?
formals : ( ⟨ <id id> ⟩^⊕ )
actuals : ( ⟨ { body } ⟩^⊕ )
```

To motivate a simple example illustrating this extension, we assume that the base language does not allow side-effects in initialization expressions. Thus, we can no longer use the `e++` expression. Instead, we inductively build an appropriate constant expression which is passed as an argument:

```
macro <decls> enum  <id I> <enums: decls Ds>({ 1 })  ; ::= {
  const int <I> = 0;
  <Ds>
}

metamorph <decls> enums(<exp E>) -->
          , <id I> <enums: decls Ds>({ <E> + 1 }) ::= {
  const int <I> = <E>;
  <Ds>
}

metamorph <decls> enums(<exp E>) --> ::= {}
```

Using this variation of the `enum` macro, we obtain the following expansion:

```
                                  const int a = 0;
                                  const int b = 1;
enum { a, b, c, d };      ⇒       const int c = 1+1;
                                  const int d = 1+1+1;
```

In more involved and ambitious applications, the arguments play the roles of "syntactic continuations".

15.8 Growing New Languages

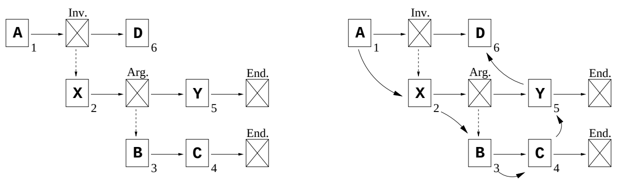
Section 15.4 contains examples that use macros to enrich the host language with new concepts and constructs. A more radical use of particularly metamorphisms is to design and implement a completely new language at very little cost.

Our host language `<bigwig>` is itself a domain-specific language designed to facilitate the implementation of interactive Web services. To program a family of highly specialized services it can be advantageous to first define what we shall call a *very* domain-specific language, or VDSL.

We consider a concrete example. At the University of Aarhus, undergraduate Computer Science students must complete a Bachelor's degree in one of several fields. The requirements that must be satisfied are surprisingly complicated. To guide students towards this goal, they must maintain a so-called "Bachelor's contract" that plans their remaining studies and discovers potential problems. This process is supported by a Web service that for each student iteratively accepts past and future course activities, checks them against all requirements, and diagnoses violations until a legal contract is composed. This service was first written as a straight `<bigwig>` application, but quickly became annoying to maintain. Thus it was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in pseudo natural language style. This makes it possible for a secretary—or even the responsible faculty member—to maintain and update the service. Figure 15.7 shows an example of the input. There is only a single macro, `studies`, which accepts as argument an entire specification in the VDSL syntax, defined using 27 metamorph rules. Its result is a corresponding `<bigwig>` service. Apart from the keyword `require`, none of the syntax shown is native to `<bigwig>`. The file `bach.wigmac` is only 400 lines and yet contains a complete implementation of the new language, including "parser" and "code generator". Thus, our macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with almost arbitrary syntax. Error trailing and unexpanded pretty printing supports the illusion that a genuinely new language is provided.

15.9 Implementation

The work presented is fully implemented in the `<bigwig>` compiler. The implementation is in `C` with extensive support from `CPP` and is available from the `<bigwig>` project homepage [74] in an Open Source distribution. In the following we present two important aspects from the implementation that achieve transparency for all other phases of the compiler. These are the *transparent*



(a) Ordinary (b) Weaved

Figure 15.4: Macro representations.

representation of macros and the *generic pretty printer* responsible for communicating macro-conscious information. These aspects support the illusion that the host language is really extended.

15.9.1 Transparent Representation

Consider the following macro definition:

```
macro <ids> xIDy ( <ids Is> ) ::= {
  X,<Is>,Y
}
```

The representation of the parse tree for the identifier list "`A,xIDy(B,C),D`" is seen in Figure 15.4(a). All node kinds of the parse tree are capable of holding three explicit macro nodes: `Inv`, `Arg`, and `End`.

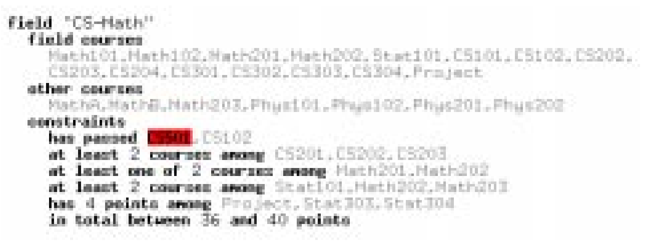
This representation yields a perfectly balanced structure with complete knowledge of the scope of all macro invocations and arguments. It is, however, clearly not transparent for subsequent phases in the compiler. Transparency is achieved through a *weaving phase* in which new pointers are after parsing short-circuited around the macro nodes giving two ways of traversing the parse tree. Macro conscious phases follow the paths in Figure 15.4(a), while macro ignorant phases only see the new short-circuited paths of Figure 15.4(b). Desugaring is not fully compatible with preserving macro information [91] and this is the only sense in which transparency is not completely achieved. However, explicit desugaring is not really necessary in a compiler that supports metamorphic syntax macros since it can be handled by the macros.

15.9.2 Generic Pretty Printing

Four *indent directives* control the pretty printing of macros:

```
param  :  \/ | \n | \+ | \-
```

The macro header is augmented with *whitespace supression*, *newline*, *indent*, and *unindent* directives. The pretty printer can be instructed to print the `si-sinon` statement without spaces around the conditional expression and with a newline before the alternate branch:



```
field "CS-Math"
  field courses
    Math101,Math102,Math201,Math202,Stat101,CS101,CS102,CS202,
    CS203,CS204,CS301,CS302,CS303,CS304,Project
  other courses
    MathA,MathB,Math203,Phys101,Phys102,Phys201,Phys202
  constraints
    has passed CS501,CS102
    at least 2 courses among CS201,CS202,CS203
    at least one of 2 courses among Math201,Math202
    at least 2 courses among Stat101,Math202,Math203
    has 4 points among Project,Stat303,Stat304
    in total between 36 and 40 points
```

Figure 15.5: HTML pretty print with an error message.

```
macro <stm> si (\/<exp E>\/) <stm S> \n sinon <stm S2> ::= { ... }
```

A more sophisticated indention correctly renders the `switch` control structure:

```
macro <stm> switch (\/<exp E>\/) { \+\n <swbody: stm S> \-\n } ...
```

These extensions are purely cosmetic; they have no semantics attached and are ignored in the invocation challenge rounds.

Our implementation supports a *generic nonterminal pretty printer* that together with a *specific terminal pretty printer* will *unparse* the code with or without macro expansion. This only depends on the choice of arrows in Figure 15.4(b).

Our implementation currently has three *terminal pretty printers* for printing `ascii`, `LaTeX`, and `HTML/JavaScript` of which the last is by far the most sophisticated. It inserts use-def hyperlinks, visualizes expression types, highlights errors, and expands individual macros at the click of a button.

15.9.3 Error Reporting

With our generic pretty printing strategy, error reporting is a special case of pretty printing using a special kind of terminal printer that only print nodes with a non-empty error string. Consequently, error messages can be viewed with or without macro expansion. Figure 15.5 shows how a simple error is pinpointed in the unexpanded syntax. The compiler can be instructed to dump the error trail as follows:

```
*** symbol errors:
*** bach.wig:175:
    Identifier ‘CS501’ not declared
      in macro argument ‘I’
      in macro invocation ‘course_ids’ (bach.wig:175) defined in [bach.wigmac:60]
      in macro argument ‘C’
      in macro invocation ‘cons’ (bach.wig:175) defined in [bach.wigmac:112]
      in macro argument ‘C’
      in macro invocation ‘cons_list’ (bach.wig:175) defined in [bach.wigmac:126]
      in macro argument ‘CN’
      in macro invocation ‘fields’ (bach.wig:168) defined in [bach.wigmac:134]
      in macro argument ‘A’
      in macro invocation ‘studies’ (bach.wig:3) defined in [bach.wigmac:158]
```

which is useful when debugging macro definitions.

15.10 Conclusion and Future Work

We have designed and implemented a safe and efficient macro language that is sufficiently powerful to grow domain-specific extensions of host languages or even entire new languages.

There are several avenues for future work. First, we will take this approach even further, by defining a notion of *invocation constraints* that restrict the possible uses of macros. Such constraints capture some aspects of the static semantic analysis of the language extensions that are grown. The constraints work exclusively on the parse tree, similarly to [30], and thus preserve transparency. Second, we will build implementations for other host languages, in particular Java. Third, it is possible to create a parser generator that given a host grammar builds a parser that automatically supports metamorphic syntax macros. Most of the required techniques are already present in the implementation of metamorphisms.

Acknowledgments

The authors thank the anonymous referees, Tommy Thorn, Eric Kidd, Peter Froehlich, Don Batory, Roger Crew, Daniel Weise, and the entire `<bigwig>` team for helpful commments and discussions.

```
macro <formula> allow <id L> when <formula F> ::= {
  all now: <L>(now) => restrict <F> by now;
}

macro <formula> forbid <id L> when <formula F> ::= {
  allow <L> when !<F>
}

macro <formula> mutex ( <id A> , <id B> ) ::= {
  forbid <A> when (is t: <A>(t) && (all s: t<s => !<B>(s)))
}

macro <toplevel> region <id R> ; ::= {
  constraint {
    label <R>~A, <R>~B;
    mutex(<R>~A, <R>~B);
  }
}

macro <stm> exclusive ( <id R> ) <stm S> ::= {
  { wait <R>~A;
    <S>
    wait <R>~B;
  }
}
```
tb
```
macro <toplevels> resource <id R> ; ::= {
  region <R>;
  constraint { ... }
}

macro <stm> reader ( <id R> ) <stm S> ::= {
  { wait <R>~enterR;
    <S>
    wait <R>~exitR;
  }
}

macro <stm> writer ( <id R> ) <stm S> ::= {
  { wait <R>~P;
    exclusive (<R>) <S>
  }
}

macro <toplevels> protected <type T> <id I> ; ::= {
  <T> <I>; resource <I>;
}
```

Figure 15.6: Concurrency control abstractions

```
require "bach.wigmac"

studies
  course Math101
    title "Mathematics 101"
    2 points fall term
  ...
  course Phys202
    title "Physics 202"
    2 points spring term
  course Lab304
    title "Lab Work 304"
    1 point fall term

  exclusions
    Math101 <> MathA
    Math102 <> MathB

  prerequisites
    Math101,Math102 < Math201,Math202,Math203,Math204
    CS101,CS102 < CS201,CS203
    Math101,CS101 < CS202
    Math101 < Stat101
    CS202,CS203 < CS301,CS302,CS303,CS304
    Phys101,Phys102 < Phys201,Phys202,Phys203,Phys301
    Phys203 < Phys302,Phys303,Lab301,Lab302,Lab303
    Lab101,Lab102 < Lab201,Lab202
    Lab201,Lab202 < Lab301,Lab302,Lab303,Lab304

  field "CS-Math"
    field courses
      Math101,Math102,Math201,Math202,Stat101,CS101,
      CS102,CS201,CS202,CS203,CS204,CS301,CS302,CS303,
      CS304,Project
    other courses
      MathA,MathB,Math203,Math204,Phys101,Phys102,
      Phys201,Phys202
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      at least one of Math201,Math202
      at least 2 courses among Stat101,Math202,Math203
      has 4 points among Project,CS303,CS304
      in total between 36 and 40 points

  field "CS-Physics"
    field courses
      MathA,MathB,Stat101,CS101,CS102,CS201,CS202,
      CS203,CS204,CS301,CS302,CS303,CS304,Project,
      Phys101,Phys102,Phys201,Lab101,Lab102,Lab201,
      Lab202
    other courses
      Phys202,Phys301,Phys302,Phys303,Phys304,Lab301,
      Lab302,Lab303,Lab304,Math202,Math203,Math204
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      has passed Phys101,Phys102
      has 4 points among MathA,MathB,Math101,Math102
      has 6 points among Phys201,Phys202,Lab101,Lab102,
                         Lab201,Lab202
      in total between 38 and 40 points
```

Figure 15.7: A VDSL for Bachelor's contracts.

Appendix

Audio/Video recorded presentations given at Microsoft Research and IBM Research:

- `<bigwig>`**—A Language for Developing Interactive Web Services**
  Available at: `http://www.brics.dk/~brabrand/bigwig-ms.asf`
  Given at Microsoft Research, Redmond, WA
  on March 20, 2000
  72min
- **Flexible, Safe, and Efficient Dynamic Generation of HTML**
  Available at: `http://www.brics.dk/~brabrand/bigwig-ibm.mpg`
  Given at IBM T. J. Watson Research Center, Hawthorne, NY
  on July 13, 2001
  62min

Bibliography

[1] V. Apparao et al. Document Object Model (DOM) level 1 specification, October 1998. W3C Recommendation. http://www.w3.org/TR/REC-DOM-Level-1/.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, June 2000.

[3] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proc. Conference on Domain-Specific Languages, DSL '97*. USENIX, October 1997.

[4] D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: a domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May/June 1999.

[5] L. Atkinson. *Core PHP Programming*. Prentice Hall, 2nd edition, August 2000.

[6] J. Bachrach and K. Playford. The Java Syntactic Extender. In *Object-Oriented Programming, Languages, and Systems (OOPSLA)*, 2001.

[7] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '02*, pages 270–281, June 2002.

[8] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web Journal*, 2(1):15–28, January 1999. Kluwer.

[9] G. Barish and K. Obraczka. World Wide Web caching: Trends and techniques. *IEEE Communications Magazine, Internet Technology Series*, 38(5):178–184, May 2000.

[10] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Fifth International Conference on Software Reuse*, 1998.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0, May 1996. RFC1945. http://www.w3.org/Protocols/rfc1945/rfc1945.

[12] P. V. Biron and A. Malhotra. XML Schema part 2: Datatypes, May 2001. W3C Recommendation. `http://www.w3.org/TR/xmlschema-2/`.

[13] C. Brabrand. Synthesizing safety controllers for interactive Web services. Master's thesis, Department of Computer Science, University of Aarhus, December 1998. Available from `http://www.brics.dk/~brabrand/thesis/`.

[14] C. Brabrand, A. Møller, S. Olesen, and M. I. Schwartzbach. Language-based caching of dynamically generated HTML. *World Wide Web Journal*, 2002. Kluwer. (See Dissertation Chapter 14).

[15] C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. PowerForms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, December 2000. Baltzer Science Publishers. (See Dissertation Chapter 12).

[16] C. Brabrand, A. Møller, A. Sandholm, and M. I. Schwartzbach. A runtime system for interactive Web services. *Computer Networks*, 31(11-16):1391–1401, May 1999. Elsevier. Also in Proc. 8th International World Wide Web Conference, WWW8. (See Dissertation Chapter 11).

[17] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 221–231, June 2001. (See Dissertation Chapter 13).

[18] C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2), 2002. (See Dissertation Chapter 10).

[19] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '02*, January 2002. (See Dissertation Chapter 15).

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (second edition), October 2000. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

[21] R. Brooks-Bilson. *Programming ColdFusion.* O'Reilly & Associates, August 2001.

[22] W. R. Campbell. A compiler definition facility based on the syntactic macro. *Computer Journal*, 21(1):35–41, 1975.

[23] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the Web. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware '98.* Springer-Verlag, September 1998.

[24] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, 1994.

[25] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic Web data. In *Proc. 18th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM '99*, March 1999.

[26] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. Technical Report RS-02-11, BRICS, March 2002.

[27] J. Clark and S. DeRose. XML path language, November 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.

[28] W. Clinger and J. Rees. Macros that work. In *Principles of Programming Languages (POPL)*, pages 155–162, 1991.

[29] K. Cox, T. Ball, and J. C. Ramming. Lunchbot: A tale of two ways to program Web services. Technical Report BL0112650-960216-06TM, AT&T Bell Laboratories, 1996.

[30] N. Damgaard, N. Klarlund, and M. I. Schwartzbach. YakYak: Parsing with logical side constraints. In G. Rozenberg and W. Thomas, editors, *Developments in Language Theory. Foundations, Applications, and Perspectives*, pages 286–304. World Scientific, November 2000.

[31] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proc. 1st USENIX Symposium on Internet Technologies and Systems, USITS '97*, December 1997.

[32] M. Dubinko, S. Schnitzenbaumer, M. Wedel, and D. Raggett. XForms requirements, April 2001. W3C Working Draft. `http://www.w3.org/TR/xhtml-forms-req.html`.

[33] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):83–110, 1993.

[34] M. F. Fernandez, D. Suciu, and I. Tatarinov. Declarative specification of data-intensive Web sites. In *Proc. 2nd Conference on Domain-Specific Languages, DSL '99*. USENIX/ACM, October 1999.

[35] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, June 1998.

[36] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0, November 1996. `http://home.netscape.com/eng/ssl3/draft302.txt`.

[37] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol, HTTP/1.1, 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[38] A. Girgensohn and A. Lee. Seamless integration of interactive forms into the Web. *Computer Networks and ISDN Systems*, 29(8-13):1531–1542, September 1997. Elsevier. Also in Proc. 6th International World Wide Web Conference, WWW6.

[39] S. Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, March 1996.

[40] M. Hanus. High-level server side web scripting in curry. In *3rd Int. Symposium on Practical Aspects of Declarative Languages, PADL'01*, pages 76–92, 2001.

[41] A. Homer, J. Schenken, M. Gibbs, J. D. Narkiewicz, J. Bell, M. Clark, A. Elmhorst, B. Lee, M. Milner, and A. Rehan. *ASP.NET Programmer's Reference*. Wrox Press, September 2001.

[42] H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Informal Proceedings of the Workshop on Programming Language Technologies for XML, PLAN-X 2002*, 2002.

[43] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB '00*, volume 1997 of *LNCS*. Springer-Verlag, May 2000.

[44] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '01*, January 2001.

[45] T. Hune and A. Sandholm. A case study on using automata in control synthesis. In *Proc. 5rd International Conference on Fundamental Approaches to Software Engineering, FASE '00*, volume 1783 of *LNCS*. Springer-Verlag, March/April 2000.

[46] ICONOCAST Inc. ICONOCAST Newsletter, August 17, 2000. `http://www.iconocast.com/issue/20000817.html`.

[47] A. Iyengar and J. Challenger. Improving Web server performance by caching dynamic data. In *Proc. 1st USENIX Symposium on Internet Technologies and Systems, USITS '97*, December 1997.

[48] R. Kelsey, W. Clinger, and J. R. (Eds.). Revised(5) report on the algorithmic language scheme (r5rs), 1998.

[49] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 1978.

[50] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[51] N. Klarlund, A. Møller, and M. I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 9(3):285–319, 2002. Kluwer. Preliminary version in Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00.

[52] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Lisp and Functional Programming*, pages 151–161, 1986.

[53] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Principles of Programming Languages (POPL)*, pages 77–84. ACM, 1987.

[54] J. Korpela. JavaScript and HTML: Possibilities and caveats, 2000. `http://www.hut.fi/u/jkorpela/forms/javascript.html`.

[55] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia services. *World Wide Web Journal*, 1(1), January 1996. O'Reilly & Associates. Proc. 4th International World Wide Web Conference, WWW4.

[56] B. M. Leavenworth. Syntax macros and extended translation. *CACM*, 1966.

[57] M. R. Levy. Web programming in guide. *Software: Practice and Experience*, 28(15):1581–1603, 1998.

[58] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. *ACM SIGMOD Record*, 29(2):153–164, June 2000.

[59] W. Maddox. Semantically-sensitive macroprocessing. Technical report, University of California, Berkeley, 1989. Technical Report UCB/CSD 89/545.

[60] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. Draft. Available from `http://www.cse.ogi.edu/~mbs/pub/xmlambda/`, 1999.

[61] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '97*, September 1997.

[62] A. Møller. MONA project home page. `http://www.brics.dk/mona/`.

[63] Netscape Corp. JavaScript form validation sample code, 1999. `http://developer.netscape.com/docs/examples/javascript/formval/overview.html`.

[64] J. Nielsen. *Designing Web Usability: The Practice of Simplicity.* New Riders Publishing, December 1999.

[65] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, October 1999.

[66] Open Market. FastCGI: A high-performance Web server interface, April 1996. Available from `http://www.fastengines.com/whitepapers/`.

[67] S. Pemberton et al. XHTML 1.0: The extensible hypertext markup language, January 2000. W3C Recommendation. `http://www.w3.org/TR/xhtml1`.

[68] D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 specification, December 1999. W3C Recommendation. `http://www.w3.org/TR/html4/`.

[69] K. Rajamani and A. Cox. A simple and effective caching scheme for dynamic content. Technical report, CS Dept., Rice University, September 2000.

[70] D. Sandberg. Lithe: A language combining a flexible syntax and classes. In *Principles of Programming Languages (POPL)*, pages 142–145, 1982.

[71] A. Sandholm and M. I. Schwartzbach. Distributed safety controllers for Web services. In *Proc. 3rd International Conference on Fundamental Approaches to Software Engineering, FASE '98*, volume 1382 of *LNCS*. Springer-Verlag, March/April 1998.

[72] A. Sandholm and M. I. Schwartzbach. A type system for dynamic Web documents. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00*, January 2000.

[73] S. Schnitzenbaumer, M. Wedel, and M. Gunatilake. XHTML-FML 1.0: Forms markup language, 1999. Stack Overflow AG. `http://www.mozquito.org/documentation/spec_xhtml-fml.html`.

[74] M. I. Schwartzbach et al. `<bigwig>` project home page. `http://www.brics.dk/bigwig/`.

[75] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of 11th ACM International World Wide Web Conference*, 2002.

[76] A. Shalit. *The Dylan Reference Manual.* Addison-Wesley-Longman, 1996.

[77] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting result equivalence in caching dynamic Web content. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[78] R. M. Stallman. The C preprocessor online documentation. `http://gcc.gnu.org/onlinedocs/cpp_toc.html`.

[79] G. Steele. Growing a language. *Lisp and Symbolic Computation*, 1998.

[80] B. Stroustrup. *The C++ Programming Language*, chapter 13. Addison Wesley, third edition, 1997.

[81] Sun Microsystems. Java Servlet Specification, Version 2.3, 2001. Available from `http://java.sun.com/products/servlet/`.

[82] Sun Microsystems. JavaServer Pages Specification, Version 1.2, 2001. Available from `http://java.sun.com/products/jsp/`.

[83] R. D. Tennent. *Principles of Programming Languages.* Prentice Hall, 1981.

[84] P. Thiemann. A typed representation for html and xml documents in haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.

[85] P. Thiemann. Wash/cgi: Server-side web scripting with sessions and typed, compositional forms. In *4th Int. Symposium on Practical Aspects of Declarative Languages, PADL'02*, 2002.

[86] P. Thistlewaite and S. Ball. Active FORMs. *Computer Networks and ISDN Systems*, 28(7-11):1355–1364, May 1996. Elsevier. Also in Proc. 5th International World Wide Web Conference, WWW5.

[87] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[88] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema part 1: Structures, May 2001. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

[89] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[90] T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1999.

[91] O. Waddell and R. K. Dybvig. Visualizing partial evaluation. In *ACM Computing Surveys Symposium on Partial Evaluation*, volume 30(3es):24-es, September 1998.

[92] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Principles of Programming Languages (POPL)*, pages 203–213, 1999.

[93] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.

[94] M. Webb and M. Plungjan. JavaScript form FAQ knowledge base, 2000. `http://developer.irt.org/script/form.htm`.

[95] D. Weise and R. F. Crew. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*, pages 156–165, 1993.

[96] D. Weise and R. F. Crew. Programmable syntax macros. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '93*, June 1993.

[97] C. Wills and M. Mikhailov. Studying the impact of more complete server information on Web caching. *Computer Communications*, 24(2):184–190, February 2001. Elsevier. Also in Proc. 5th International Web Caching and Content Delivery Workshop.

[98] A. Wolman. Characterizing Web workloads to improve performance, July 2000. University of Washington. Available from `http://www.cs.washington.edu/homes/wolman/generals/`.

[99] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive Web sites. In *Proc. 26th International Conference on Very Large Data Bases, VLDB '2000*. Morgan Kaufmann, September 2000.

[100] H. Zhu and T. Yang. Class-based cache management for dynamic Web contents. In *Proc. 20th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM '01*, pages 1215–1224, April 2001.