Caching for XML Generators: An Application of Two-Level Techniques

Peter Thiemann
Universität Freiburg
Germany
thiemann@informatik.uni-freiburg.de

Abstract

Caching of documents is an important concern on the Web. It is a major win in all situations where bandwidth is limited. Unfortunately, the increasing spread of dynamically generated documents seriously hampers traditional caching techniques in browsers and on proxy servers.

WASH/CGI is a Haskell-based domain specific language for creating interactive Web applications. The Web pages generated by a WASH/CGI application are highly dynamic and cannot be cached with traditional means.

We show how to implement the dynamic caching scheme of the BigWig language [2] in the context of WASH/CGI. The main issue in BigWig's caching scheme is the distinction between fixed parts (that should be cached) and variable parts (that need not be cached) of a document. Since BigWig is a standalone domain-specific language, its compiler can perform the distinction as part of its static analysis. Hence, the challenge in our implementation is to obtain the same information without involving the compiler. To this end, we extend WASH/CGI's document language by mode annotations and define the translation of the resulting annotated document language into JavaScript.

To alleviate the awkwardness of programming directly in annotated language, we have defined a surface syntax in the style of HSP (Haskell Server Pages) [11].

Keywords

Web Programming, Caching, Annotated Languages

1 Introduction

The lack of bandwidth is a major problem that hampers the effective use of Web applications. In the early days of the Web, when bandwidth was really scarce, the standard solution was to employ caching. Hierarchical networks of caching proxy servers were established in order to share bandwidth as much as possible.

However, the content of the Web has also changed along with its growth. While the majority of documents in the early Web were static, today's Web pages are highly dynamic. They are generated from document templates depending on user profiles, language and image preferences, and other database operations on the server. With the advent of these individual, personalized Web pages, the usual caching schemes are easily defeated. Unfortunately, this has become true for the majority of the pages because the percentage of dynamically generated pages is fast increasing, as a recent survey indicates [1].

For these reasons, every Web programming environment that can create dynamic content should also provide a caching strategy that alleviates the load of the network. Many schemes work a-posteriori by employing designated cache server proxies that cache dynamic documents and obtain information about the freshness of the cached copies from objects and object dependencies in the underlying database (*e.g.*, [4, 5]). Other systems suggest the use of a macro language to compose dynamic documents. This approach also requires specialized proxies that expand these macros or the installation of specialized browsers or plugins (*e.g.*, [7]).

Further techniques that have been used are delta caching [12], which relies on the observation that many dynamically generated documents only differ in reasonably small, fixed parts, and approximate matching [14], which again requires application programmers to provide hints for cache-control.

Our proposal builds on BigWig's caching strategy, which relies completely on existing caches in proxy servers and browsers. It does not require hints from the application programmer, nor does it require the installation of specialized software on clients. However, it relies on using the BigWig language and in particular BigWig's document representation, as we will describe in detail in Section 2.

While the BigWig approach is highly effective, it's implementation really hinges on the domain-specific BigWig compiler. In the present work, we show how to exploit the BigWig caching strategy for dynamic documents without a domain-specific compiler. To substantiate our claim, the author's WASH system (a Web authoring system in Haskell) has been extended to implement cached dynamic documents.

The main tool that we are using is an annotated version of the document language where the annotations express the degree of variability. Since writing such annotations is cumbersome (and should

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

not be expected from an application programmer) we have defined an XML-style surface language for documents that should appeal to Web programmers and at the same time provides the essential information for computing the annotations. A simple preprocessor translates this surface syntax into the annotated document language.

In the remainder of the paper, we first introduce BigWig's caching strategy in Section 2. Then, in Section 3, we introduce our annotated document language. Section 4 defines its translation to (abstract) JavaScript. Section 5 defines the surface syntax and Section 6 shows its translation to an annotated document. Section 7 discusses some more related work on extending a programming language with XML notation and Section 8 concludes.

2 Caching a la BigWig

BigWig [3] is a domain specific language for building Web applications. One of its features is support for the generation of correct (X)HTML output. To this end, BigWig has a built-in abstract data type for document templates and provides markup style syntax for values of this data types. A document template represents a parameterized HTML document. Parameterization is expressed by placing named holes in the document. For example, the code

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

defines a document template `Doc` that represents a fragment of a document consisting of a text node ("You are visitor number ") and a `b` element. The contents of the `b` element is a hole named number: `<[number]>`.

Document templates admit a plug operation that specifies the name of a hole and and either a document or a string to fill the hole. The document may be a template, again, so the result of the plug operation may be a template, too. For example, in the code

```
exit Doc <[number=counter];
```

the `number` hole in `Doc` is filled with the value of variable `counter`.

Internally, a document template is represented by a vector and a tree structure with the usual nodes for representing HTML, element nodes, text nodes, and attributes nodes. In addition, there are hole nodes that point to entries in the vector (which is attached to the root of the template). The vector contains the plugged values for all the holes.

This document representation allows the efficient implementation of the plug operation and the print operation. It also enables the implementation of an interesting caching strategy. This strategy enables the use of the browser's cache for displaying dynamic documents.

The idea of BigWig's document caching strategy [2] is simple but very effective:

- Transform each document template into a JavaScript function that takes the plugged values as parameters. The function prints its standard nodes and reverts to the parameters to print the plugged values.
- Save all functions in one application-specific file and store it on the Web server.
- In the document source, the result of a plug operation is translated to the respective call of a JavaScript function.
- Since each function is imported from the application specific file, the Web browser (or a caching proxy) can download this file once and for all. All later uses of the same application only need to transmit the dynamic content of Web pages generated with BigWig.

This simple strategy decreases the amount of data transmitted from server to client by a factor between 4 and 37, depending on the particular example [2].

Clearly, such a caching strategy is a worthwhile addition to any Web programming system. Unfortunately, BigWig's implementation relies heavily on the particular document representation and on the custom compiler (BigWig is a standalone or closed domain specific language with its own compiler).

In the following, we show how to add BigWig's strategy to the WASH system by exploiting ideas from program generation (annotated languages) and using a simple textual preprocessor.

3 Annotated Document Language

An XML document is a tree with several kinds of nodes. For our purposes, the interesting ones are:

- text nodes,
- attribute nodes, and
- element nodes.

The nodes have different types and children:

```
type Tag = String
data Node =
    -- |element node
    El Tag [Attribute] [Node]
    -- |text node
  | Tx String
type AttributeName = String
type AttributeValue = String
data Attribute =
    -- |attribute node
    At AttributeName AttributeValue
```

The element constructor, `El`, takes a tag name, a list of attribute nodes, and a list of nodes to produce a new element node. The text constructor, `Tx`, constructs a text node from a string. The attribute constructor takes two strings, the attribute name and its value, as arguments.

In two-level programming languages, an annotation determines the binding time of each language construct. For our application, we distinguish between fixed nodes, which are unlikely to change, and variable nodes, which are expected to change. In addition, there is an annotation to indicate the start of a fixed document template. Unlike to two-level languages, the annotations may be arbitrarily mixed.

```
data Mode =
    S   -- start of a fixed template
  | F   -- inside fixed template (static)
  | V   -- variable (dynamic)
data Node2 =
    El2 BTime Tag [Attribute2] [Node2]
  | Tx2 BTime String
```

```
data Attribute2 =
    At2 BTime AttributeName AttrValue2
data AttrValue2 =
    Av2 BTime AttributeValue
```

An element node or a text node just has one binding time, wheres an attribute node has one binding time for the attribute name and another for the value. This is a sensible differentiation because attribute names are mostly fixed whereas attribute values may sometimes be variable.

4 Translation to JavaScript

The main step towards the implementation of caching is the translation of an annotated document into JavaScript. Conceptually, the translation has two steps. The first step maps the document into an internal structure with abstract function definitions. The second step translates these definitions into JavaScript syntax.

The following datatype represents a subset of JavaScript function definitions.

```
data FunDef =  FunDef { fname :: String
                      , fargs :: [String]
                      , fbody :: Body }
type Body = [Expression]
data Expression =
     Output String
   | FCall  String [[Expression]]
   | FVar   String
```

A function definition consists of a function name, `fname`, a list of formal parameter names, `fargs`, and a function body, `fbody`. The body is a list of expressions, which are executed for side effect. An expression can either output a string, call a function, or output the value of a variable. A function call is characterized by the name of the function and the parameter list. In the parameter list, the datatype `Expression` is reused with a slightly different meaning. Each parameter is the value of a *list* of `Expression`, where the values of the single expressions are concatenated. That is,

```
FCall "f1" [[Output "<b>", FVar "v2", Output "</b>"]]
```

represents a call to a unary function `f1`. The single argument is the concatenation of `"<b>"`, the value of `v2`, and `"</b>"`. Nested function calls do not occur nested inside parameter lists.

The idea of the translation is to collect the variable parts and generate parameterized code for the fixed parts. The variable parts are propagated to the toplevel. Thus, the result of translating a document template is a list of function definitions (for the fixed parts) and a function call which takes all variable parts as parameters. For readers acquainted with two-level languages, this puts the usual world order upside-down.

The translation relies on a monad `TM` that encapsulates name generation, collection of function definitions, and output of a list of `Expressions`. That is, it is a combination of a state transformer (for name generation and functions) and a writer monad. Instead of presenting the standard definition for this monad, we show the signatures of the special operations that our translation uses.

```
-- |creates a new variable name
newVar :: TM String
-- |creates a new function definitions from a list
-- of formal parameter names and a body
newFunDef :: [String] -> Body -> TM String
-- |generate an output expression
outString :: String -> TM ()
-- |generate a function call expression
outCall :: String -> [[Expression]] -> TM ()
-- |generate a variable expression
outVar :: String -> TM ()
-- |obtains the output generated by the argument
-- computation. Generates no output.
getOut :: TM a -> TM (a, [Expression])
```

The translation works one in of three modes, `S`, `F` and `V`, corresponding to the binding time of the parent node. The mode `S` indicates that the node is the start of a fixed template. Hence, `S` implies `F`, which indicates that the node is part of a fixed template. Mode `V` indicates that the node is variable. It turns out that the interesting points in the translation are the transitions between fixed (`S` or `F`) parent and variable child and between any mode parent and `S` child (since a new template starts at this point).

In the remaining cases, when both parent and child have the same mode, the translation just generates the appropriate output.

```
trNode' :: Node2 -> TM [(String, Expression)]
trNode' (El2 mode t attrs nodes) =
  do outString ('<' : t)
     alists1 <- mapM (trAttr mode) attrs
     outString ">"
     alists2 <- mapM (trNode mode) nodes
     outString ("</" ++ t ++ ">")
     return (concat (alists1 ++ alists2))

trNode' (Tx2 mode s) =
  do outString s
     return []
```

The result of the translation is an association list. It collects the variable parts encountered in the child nodes in form of bindings to formal parameter names. The translation of an element node generates the appropriate string output and recursively outputs the attributes and the child nodes using `trAttr mode` and `trNode mode`, respectively. Each of these may generate bindings and the translation of the node just collects them. The translation of a text node cannot generate bindings.

The transition cases are more interesting. We only discuss the code for dealing with elements, the functions that deal with attributes are very similar. Elements (and text nodes) are handled by

```
trNode :: BTime -> Node2 -> TM [(String, Expression)]
```

where the first argument is the mode of the parent node and the second argument is the current node. The first case handles the transition from a fixed (`S` or `F`) parent to a variable child.

```
trNode parentmode node
  | fixed parentmode && mode node == V =
  do (_, body) <- getOut (trNode' node)
     makeAlist (contract body)
```

Since the child node is variable, it does not generate an association list so the result of the recursive call to `trNode'` is ignored. The `getOut` function reifies the output generated by the translation of the child node in variable `body`. It is easy to see that (for a variable node) the output list consists solely of string output `Output s` and functions calls, where the arguments are again string output.

On this body, the function contract first performs some simplification, and then makeAlist constructs an association list and outputs a list of variable references, FVar, and function calls, FCall. The invariant of makeAlist is that substituting for the variables in the output according to the association list yields an output equivalent to body. Both functions are easy list programming exercises, but we've put them into the appendix for reference.

The next case deals with the transition between any node and the start of a template.

```
trNode m node | mode node == S =
  do (alist, body) <- getOut (trNode' node)
     f <- newFunDef (map fst alist) (contract body)
     makeAlist [FCall f (map (return . snd) alist)]
```

It also starts by obtaining the output generated in the recursive call. Since that node is guaranteed to be fixed, it may compute a non-empty association list. From this information, newFunDef constructs a new function definition, taking the domain of the association list serves as formal parameters and the body list as the function's body. Finally, makeAlist generates a call to the new function with all previous strings as parameters.

In all other cases, the code just falls through to trNode:

```
trNode m node =
  trNode' node
```

The final step, the translation to actual JavaScript syntax is straightforward and hence omitted from the paper. Instead, we give two examples of (small) translated documents.

The first document demonstrates a fixed template with variable text in it. The second document is a larger template with a variable attribute value deep inside.

```
ex1 = El2 S "h1" [] [Tx2 V "some text"]
ex2 = El2 S "html" []
        [ El2 F "head" [] [El2 F "title" [] [el2]]
        , El2 F "body" []
           [ex1
           ,El2 F "a"
             [At2 F "href"
               (Av2 V "mailto:thiemann@acm.org")]
             [Tx2 V "send me mail!"]]]
```

For ex1 the translation yields

```
function f2 (out, v1) {
  f1(out, v1);
}

function f1 (out, v1) {
  out.write("<h1>");
  out.write(v1);
  out.write("</h1>");
}

  f2(out, "some text");
```

and ex2 (which contains ex1) translates to

```
function f3 (out, v1, v2, v3, v4) {
  f2(out, v1, v2, v3, v4);
}

function f2 (out, v1, v2, v3, v4) {
  out.write("<html><head><title>");
  out.write(v1);
  out.write("</title></head><body>");
  f1(out, v2);
  out.write("<a href=");
  out.write(v3);
  out.write(">");
  out.write(v4);
  out.write("</a></body></html>");
}

function f1 (out, v1) {
  out.write("<h1>");
  out.write(v1);
  out.write("</h1>");
}

  f3(out, "more text", "some text",
          "\"mailto:thiemann@acm.org\"",
          "send me mail!");
```

The key to make the caching scheme work is as follows.

- Each function f is stored in an application-specific like-named JavaScript source file *app_f*.js on the server (one function per file).
- The actually generated Web page just loads all referenced JavaScript source files, sets the output object, and calls the main function.

The two example elements above generate four distinct function definitions/files, one function corresponding to the ex1 element is shared between the two documents. In the first case, the main document would contain

```
<script src="ex1_f2.js"> </script>
<script src="ex1_f1.js"> </script>
<script>
f2(document, "some text");
</script>
```

In the second case, the main document would contain

```
<script src="ex2_f3.js"> </script>
<script src="ex2_f2.js"> </script>
<script src="ex2_f1.js"> </script>
<script>
f3(document, "more text", "some text",
             "\"mailto:thiemann@acm.org\"",
             "send me mail!");
</script>
```

The full naming scheme must, of course, avoid name clashes among the generated function names and file names. In this paper, we are using a simplified version to increase the readability of the examples.

5 Surface Syntax

Generating XHTML with the WASH system [15, 16] is based on a document monad WithHTML m a. A value in this monad stands for a sequence of document nodes. Each kind of nodes has a particular constructor:

- a text node is constructed from a text string by `text :: String -> WithHTML m a`;

- an element node with tag `t` is constructed by the identically named `t :: WithHTML m a -> WithHTML m a`; and
- an attribute node is constructed from two strings, the attribute name and its value, by `attr :: String -> String -> WithHTML m a`.

Each of the constructors creates a singleton sequence containing just the new node. The standard monadic `>>=` operator (in a few variations) serves as sequence concatenation and the monadic `return ()` operation creates an empty sequence (also denoted by `empty`). In particular, the `do` notation can be employed to create and concatenate sequences.

Due to the embedding of WASH in Haskell, no notion of document templates (as in the BigWig system) is required. A document template is simply a Haskell function with parameters of type `WithHTML m a`. The following example function from a WASH program demonstrates this fact. Given a multiplier `mpy`, the function `ptable` creates a multiplication table for the numbers 1 to 12.

```
ptable :: Int -> CGI ()
ptable mpy =
  standardQuery "Multiplication Table" $
  table (mapM_ pLine [1..12])
  where
    align = attr "align" "right"
    pLine i =
      tr (do td (text (show i) ## align)
             td (text "*")
             td (text (show mpy))
             td (text "=")
             td (text (show (i * mpy)) ## align))
```

Clearly, this function is a concise implementation of the task at hand. However, it is virtually impossible (save perhaps with an expensive static analysis in a custom compiler) to extract fixed patterns from that code because there is no special notion of document templates.

Of course, a WASH programmer could be asked to program in an annotated version of the language, modeled after the `Node2` datatype from Section 3. In this language, the templates would be readily apparent. However, we rejected this idea because of its awkwardness.

Instead, we adopted a surface syntax that makes fixed patterns explicit without sacrificing WASH's flexibility. Haskell Server Pages [11][1] served as the starting point for developing that surface syntax. Before delving into a description of this syntax, here is the transcription of the above multiplication table to WASH syntax.

```
ptable :: Int -> CGI ()
ptable mpy =
  standardQuery "Multiplication Table" $
  <table> <% mapM_ pLine [1..12] %> </table>
  where
    align = <[ align="right" ]>
    pLine i =
      <tr><td><% text (show i) %><% align %></td>
          <td>*</td>
          <td><% text (show mpy) %></td>
          <td>=</td>
          <td><% text (show (i * mpy)) %>
              <% align %></td>
      </tr>
```

[1] Thanks to Erik Meijer and Danny van Velzen for making the code for their preprocessor available to the author.

The syntax is a simple extension of Haskell's syntax by XML-style markup. Like Haskell Server Pages (and unlike Java Server Pages), it emphasizes compositionality. Hence, the syntax extension is completely described by six simple rules.

1. Wherever a WASH program expects a node sequence, a program can use
   - standard XML element syntax to build a sequence containing that one element node;
   - `<[`*`attributeName`*`="`*`attributeValue`*`"]>` to create (a sequence with) just one attribute node;
   - the notation `<#>`*`XMLcontent`*`</#>` to create any *sequence* of nodes as indicated by *`XMLcontent`*.
2. Inside an XML element's body, a document sequence (an expression of type `WithHTML m a`) can be inserted by wrapping it into escape brackets: `<%` *`documentSequence`* `%>`.
3. In place of a literal value for an XML attribute, any string value can be inserted by `<%` *`stringValue`* `%>`.
4. Escaped document sequence expressions may be arbitrarily nested: They can contain any of the alternatives from rule 1.

For example, in

```
helo5 = standardQuery "What's your name?" $
  <#>Hi there! What's your name? <%
     activate greeting textInputField empty %>
  </#>
```

the `<#>` notation must be used because the constructed XHTML fragment has (at least) two nodes, a text node and an `input` element node. The `<#>` notation also provides a different notation for `empty`:

```
empty = <#> </#>
```

As a final example for a code block instead of an attribute value consider the function

```
alignment :: String -> WithHTML x a
alignment what = <[align=<% what %>]>
```

which constructs an `align` attribute.

There is a slight caveat in using the markup syntax. The `text` constructor for text nodes automatically translates illegal characters like `<`, `&`, etc to their XHTML entity references. Using the markup notation, the XML conventions have to be observed in the source because the text is passed on verbatim without further processing.

The astute reader may miss syntax corresponding to an attribute hole, as in the use of `at` in this example:

```
let at = attr "href" "http://www.w3c.org" in
-- THIS IS ILLEGAL:
<a <% at %>>The W3C</a>
```

However, this functionality is not required because WASH document sequences also collect the attributes for the *enclosing* element. That is, the above element should be programmed as

```
let at = attr "href" "http://www.w3c.org" in
```

```
-- THIS IS CORRECT:
<a><% at %>The W3C</a>
```

and the same result would be achieved by placing the escape anywhere else inside the a element.

```
<a>The W3C<% at %></a>
```

6 Translation to Annotated Syntax

The translation to annotated document syntax is implemented in the WASH preprocessor. The idea is fairly straightforward.

All standard document constructors (element constructors like a, head, body, ...; or the attribute constructor attr) create variable nodes. In addition, there are annotated version available (also to the programmer) to construct nodes in either of the fixed modes, S and F. For example, there are a_S, a_F, head_S, head_F, attr_S, and so on. Of course, the programmer can use these directly, but the preprocessor can do it very easily as follow.

The preprocessor "parses" the Haskell parts of the source only very superficially. Essentially, it searches for the beginning of an XML markup. Whenever it finds such markup, it switches to XML parsing and translates an element, content sequence (in case of <#>), or attribute. In the translation, the toplevel node of each markup is annotated with S (start of fixed template) and all nodes deeper down are annotated with F (fixed).

For example, the translation of the above multiplication table example looks like this (after inserting line breaks and indentation for readability in the definition of pLine):

```
ptable :: Int -> CGI ()
ptable mpy =
  standardQuery "Multiplication Table" $
  table_S (( mapM_ pLine [1..12] ))
  where
    align = (attr_SF "align" "right")
    pLine i =
      tr_S (
        td_F (( text (show i) ) >> ( align )) >>
        td_F (rawtext_F "*") >>
        td_F (( text (show mpy) )) >>
        td_F (rawtext_F "=") >>
        td_F (( text (show (i * mpy)) ) >> ( align )))
```

A nice, but perhaps surprising, property of the annotation is that it cannot be wrong. Mixing up the different modes does *not* lead to an error because they only control the caching behavior of the generated output. Hence, the generated output is always usable, but it may not be cached as well as it could.

7 Related Work

In the introduction, we have already commented on some related work on caching of dynamic Web documents. Here, we focus on discussing some systems that extend the syntax of a host language to provide for convenient ways of creating XML structures or generating XML output. We are not considering dedicated XML languages in this discussion.

Haskell Server Pages (HSP) [11] form the main source of inspiration for our surface syntax. However, the underlying motivation for the surface syntax is quite different. Whereas HSP aims at providing a notion similar to Java Server Pages [13] (but in a compositional way) in the context of Haskell, we employ the XML syntax to build document fragments and templates inside of a program. This difference comes from the fact that HSP supports the page-centric approach to Web programming (where generated pages and scripts are almost in one-to-one relationship) whereas WASH supports the session-centric approach where many pages may be generated inside of one source program.

SMLServer [8] also allows the embedding of dynamically generated XML in its output. However, this facility is entirely string-based and relies on a string-quoting convention supporting by some ML systems.

BigWig [3] and its successor, JWig [6], both support a built-in document datatype. This datatype is supported by a custom compiler, which also implements the caching strategy explained in Section 2.

Recently, there have been a few extensions of Java with built-in XML syntax and datatypes [10, 9], but none of them considers support for caching. In fact, both map XML internally to objects which are serialized on demand.

8 Conclusion

When the present author first learned about BigWig's caching scheme, it seemed impossible to adapt it for use in the WASH system. Apparently, too much of it relied on the specific document data structure used in that system and too much work was delegated to the domain-specific BigWig compiler.

It turned out as a surprise that a simple solution, based on an annotated document language, could be found. The need to use an annotated document language also found a satisfactory solution in the definition of the surface language. Due to its conceptual simplicity, the solution should work beyond Haskell.

References

[1] Greg Barish and Katia Obraczka. World Wide Web caching: Trends and techniques. *IEEE Communications Magazine Internet Technology Series*, May 2000.

[2] Claus Brabrand, Anders Møller, Steffan Olesen, and Michael I. Schwartzbach. Language-based caching of dynamically generated HTML. *World Wide Web Journal*, 2003.

[3] Claus Brabrand, Anders Møller, and Michael Schwartzbach. The <bigwig> Project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[4] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of the 1998 Middleware conference*, 1998.

[5] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable system for con-sistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, March 1999.

[6] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending java for high-level Web service construction. *toplas*, 2003. To appear.

[7] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, December 1997.

[8] Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *Practical Aspects of Declarative Languages, Proceedings of the Fifth International Symposium, PADL'03*, number 2562 in Lecture Notes in Computer Science, pages 74–91, New Orleans, LA, USA, January 2003. Springer-Verlag.

[9] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: An XML programming language for web service specification and composition. In *WWW 2002, International World Wide Web Conference*, Honolulu, HI, USA, may 2002.

[10] Martin Kempa and Volker Linnemann. Type checking in XOBE. In E. Rahm G. Weikum, H. Schöning, editor, *Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung*, pages 265–274, Leipzip, 2003.

[11] Erik Meijer and Danny van Velzen. Haskell Server Pages, functional programming and the battle for the middle tier. In *Draft proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, pages 23–33, Montreal, Canada, September 2000.

[12] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM*, pages 181–194, 1997.

[13] Eduardo Peligrí-Llopart and Larry Cable. Java Server Pages Specification. `http://java.sun.com/products/jsp/index.html`, 1999.

[14] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu. Exploiting result equivalence in caching dynamic Web content. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[15] Peter Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages, Proceedings of the Fourth International Workshop, PADL'02*, number 2257 in Lecture Notes in Computer Science, pages 192–208, Portland, OR, USA, January 2002. Springer-Verlag.

[16] Web authoring system in Haskell (WASH). `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH`, March 2001.

A contract, makeAlist

```
contract :: [Expression] -> [Expression]
contract (Output s: Output t: xs) =
  contract (Output (s ++ t): xs)
contract (FCall s args: xs) =
  FCall s (map contract args): contract xs
contract (x: xs) =
  x: contract xs
contract [] =
  []

makeAlist :: [Expression]
          -> TM [(String, Expression)]
makeAlist exprs =
  f [] exprs
  where
  f acc [] =
    return acc
  f acc (expr:exprs) =
    case expr of
      Output s ->
        do v <- newVar
           outVar v
           f ((v, Output s): acc) exprs
      FCall s a ->
        do listof_alist_args <-
             mapM (getOut . makeAlist) a
           let (alists, args) =
             unzip listof_alist_args
           outCall s args
           f (concat alists ++ acc) exprs
      FVar s ->
        do outVar s
           f ((s, FVar s): acc) exprs
```