Haskell Server Pages
Functional Programming and the Battle for the Middle Tier

Erik Meijer & Danny van Velzen
Utrecht University
`mailto:erik@meijcrosoft.com`
`mailto:danny@shopmontages.nl`
`http://www.cs.uu.nl/~erik`

Abstract

Haskell Server Pages (HSP) provide an easy way to create dynamic web pages and simplify the task of building middle tier components. This article gives an overview of HSP from a programmer's perspective. It includes examples of HSP in action and gives a precise description of translating HSP scripts into plain Haskell.

1 Overview

Over the next few years, virtually every major application will be completely web-based, or at a minimum have a web front end. Such distributed applications are often structured using a three-tier model, which consists of:

- *User-Services Tier:* The front-end client that communicates with the user through some user interface, eg textual, graphical, telephone- or voice-driven.
- *Business-Services Tier:* A collection of software components that encode business logic, and process the information flow between the user- and data-services tiers.
- *Data-Services Tier:* The back-end database server that stores business data.

From an application programmer's point of view all the action is in encoding the business logic and providing the glue between the user-services tier and the data-services tier.

The business-services or *middle tier* is usually built from software components that live in a special runtime environment (such as COM+, EJB, or Corba Component Services) [29] that provides services for:

- Distributed transaction management.
- Security services to control object creation and use.
- Management and pooling of processes and threads, object instances, database connections, and other resources.

Internet companies such as eBay, Amazon, Barnes and Noble, and AOL must have their business up and running 24 hours a day, 365 days a year. For instance when the eBay site went down for 48 hours recently, they lost over five million dollars.

Viewed in this light it is remarkable that middle tier components are written using weakly typed languages such as Visual Basic [28], JavaScript [4] or Perl [13]. Such languages provide no guarantees that programs don't crash or give incorrect answers. Server redundancy, process protection etc. are just band-aid measures that don't solve the real problem.

Our research program is targeted at providing languages, tools and libraries for fast and safe construction of middle tier components. To communicate between the middle tier and the data access layer we have developed *HaskellDB* [15]. To make effective use of the middle tier runtime we have added support for accessing classic COM [14] and Java [22] to Haskell, and we are currently building a new scripting language called Mondrian [25] and a new back end for GHC that compiles to Java and the new Microsoft .NET runtime.

To communicate between the presentation layer and the middle tier we have implemented *HaskellScript* [23] for writing client-side DHTML scripts and *HaskellCGI* [21], a library for writing server-side CGI scripts. Experience with the CGI library however, showed that embedding HTML or XML fragments using combinators in Haskell is rather awkward. The reason is that documents often contain more plain text than markup elements. Writing long string literals in Haskell using quotes and literal gaps[1] is not the most convenient way to denote string content in XML fragments.

In this paper we present *Haskell Server Pages* (HSP) as a new way to generate dynamic content using Haskell. The most important contributions of HSP are that

- *XML fragments are first class values:* XML fragments have the same status as ordinary Haskell expressions, so we can put them in lists, pass them as arguments to and return them as results from functions, etc. This allows many common document processing patterns to be captured in small highly reusable libraries [11].
- *XML fragments can be pattern-matched:* Constructing XML fragments is just one side of the coin. In many cases we need to transform one XML document into another. Being able to do this in a higher-order, polymorphic functional language is much more convenient than

[1] A little know feature of the Haskell language, formed by two backslashes enclosing white space. Literal gaps allow long strings to be spread across different lines.

using separate languages such as XSL [35] or frameworks like the Document Object Model (DOM) [6].

- *PCDATA fragments need not be quoted:* Since we write XML fragments using *concrete* XML syntax, PCDATA strings need not be quoted. This simplifies the construction of large documents.
- *Haskell values can be recursively embedded inside XML fragments:* We can escape from the XML level back to Haskell and embed any Haskell that is an instance of the class `IsChild` (or for attribute `IsValue`). Since `IO` is an instance of these classes, it is even possible to embed commands inside documents.
- *HSP is easily translated into pure Haskell:* As we show in this paper, HSP can be implemented as a simple preprocessor to Haskell. The amount of syntactic sugar is comparable to that of the `do`-notation or list-comprehensions, and is defined by just a handful of rules.

A popular route to implementing domain-specific languages is by embedding into Haskell [10], and HSP is no exception. However, a necessary condition for this to work is that all the features of the domain-specific language can be mapped to a corresponding feature of Haskell. In the case of HSP, Haskell falls short in two areas [24, 31]:

- *Proper pattern-matching on XML fragments requires the use of a Glushkov automaton to direct the matching of regular expressions.* Because we are piggybacking on Haskell's more primitive pattern-matching mechanisms, we are forced to introduce some over specification in certain places.
- *Guaranteeing validity of XML documents against their DTDs requires a sophisticated type system that supports type indexed rows.* HSP does guarantee well-formedness of XML fragments, but unfortunately the type system needed for validating XML cannot be simulated in Haskell.

Even well-formedness and simple pattern-matching on documents alone is already a much stronger property than most other competing languages can offer, but there are many occasions where we do want to match and type arbitrary XML. For this we are developing a new language XMLambda [24] which has XML documents as its basic data types.

In the mean time, HSP remains an excellent example of the Pareto principle[2]; we get 80 percent of the functionality of XMLambda for 20 percent of the cost.

In the rest of this paper we will first give an overview of the current state of affairs in generating documents in the middle tier (section 2) and explain why they fall short. Next we give an informal introduction to HSP (section 3) and show in detail how HSP scripts can be translated into pure Haskell (section 4). We end the paper by describing how HSP applications are integrated into the middle tier runtime environment (section 5).

[2] Dr. Joseph Juran (of total quality management fame) formulated the Pareto Principle after expanding on the work of Wilfredo Pareto, a nineteenth century economist and sociologist. It is a shorthand name for the phenomenon that in any population which contributes to a common effect, a relative few of the contributors account for the bulk of the effect

2 Web-based applications

In most 3-tier applications the work of the middle tier is usually split recursively into three tiers again. Client applications run inside a browser and submit requests to a web server using HTTP. The 'presentation layer' on the server transforms the request and passes it on to the 'business layer' which will perform some computation by interacting with the 'data layer'. The results from the 'business layer' are then transformed into HTML or XML by the 'presentation layer' and returned as the response to the client.

2.1 Server Pages

The most popular way of generating HTML (or XML) responses in the middle tier is by using so called *server pages.* A server page is a special HTML page that contains embedded scripts. Server pages exist for many languages, such as *JSP* (Java Server Pages) [26], *PSP* (Python Server Pages) [1], *PHP* (PHP: Hypertext Preprocessor) [17], *XSP* (Extensible Server Pages) [20], and *ASP* (Active Server Pages) [9]. The latter is quite interesting as it is a language independent framework. Any scripting language that supports the ActiveX Scripting Architecture can be used to write ASP scripts [16, 3].

Here is the inevitable "Hello World" example written as a server page in Visual Basic. The `Response.Write "Hello World!"` statement writes the string `"Hello World"` on the `Response` output stream that is sent to the client:

```
<HTML>
  <HEAD>
    <TITLE>Example</TITLE>
  </HEAD>
  <BODY>
      <% Response.Write "Hello World!" %>
  </BODY>
</HTML>
```

Sometimes `<%=`*e*`%>` is used as an abbreviation for `<% Response.Write` *e* `%>`. The choice of primitive functions and the tags that delimit code differ for the various flavors of server pages. For example PHP uses `echo` instead of `Response.Write`, and `<?php` and `?>` as tags but also supports `<%` and `%>`.

Server pages are typically implemented using a process called *page compilation*, which transforms a hybrid HTML/script page into a single script that will generate the required HTML. For instance, the example above would be translated into a script that writes each line of the original server page to the `Response` stream:

```
Response.Write "<HTML>"
Response.Write "<HEAD>"
Response.Write "<TITLE>Example</TITLE>"
Response.Write "</HEAD>"
Response.Write "<BODY>"
Response.Write "Hello World!"
Response.Write "</BODY>"
Response.Write "</HTML>"
```

It is important to note that the translation is not applied recursively inside code fragments.

This simple semantics can make server pages rather confusing, especially if control structures are used to generate HTML. As the next example shows, code fragments embedded inside `<%` and `%>` need not be syntactically complete phrases themselves:

```
<HTML>
  <HEAD>
    <TITLE>Example</TITLE>
  </HEAD>
  <BODY>
      <% If Domain = "nl" %>
        Hallo Wereld!
      <% Else %>
        Hello World!
      <% End If %>
  </BODY>
</HTML>
```

Server pages force you to think backwards from the fully explicit generated code to some level of HTML mixed with script to make sure that the generated code is syntactically correct:

```
Response.Write "<HTML>"
Response.Write "<HEAD>"
Response.Write "<TITLE>Example</TITLE>"
Response.Write "</HEAD>"
Response.Write "<BODY>"
If Domain = "nl"
  Response.Write "Hallo Wereld!"
Else
  Response.Write "Hello World!"
End If
Response.Write "<BODY>"
Response.Write "</HTML>"
```

2.2 Server Pages break the principle of Abstraction

The semantics of server pages is not compositional, that is, to understand what a server page means, we need to do a complete page translation and inspect the resulting code. Naive server pages provide just a very thin layer of veneer that hides a few calls to `Response.Write`, but they do not make HTML fragments into first class citizens. Hence they break Tennent's principle of abstraction [32, 27] that says that values of a syntactically relevant domain can be given a name. An abstraction mechanism that does not satisfy this basic principle is rather useless!

As an example server page that shows the lack of abstraction, we will generate an HTML table of 16x16 cells, where the background color of each cell is determined by the cell's coordinates via the (unspecified) function `GenColor`.

The ASP page contains two nested loops, the outer one generates the table rows and the inner generates the cells:

```
<TABLE border="1">
  <% For x = 1 To 16 %>
    <TR>
      <% For y = 1 To 16 %>
        <TD bgcolor=<%= genColor(x,y) %>>
          (<%= x %>,<%= y %>)
        </TD>
      <% Next %>
    </TR>
  <% Next %>
</TABLE>
```

Note that it is possible to use script inside attributes as well.

Suppose we want to lift out the inner loop into a separate procedure `GenData` to make the page more orderly.

```
<TABLE border="1">
  <% For x = 1 To 16 %>
    <TR>
      <% Call GenData (x) %>
    </TR>
  <% Next %>
</TABLE>
```

To do so, we cannot merely copy and paste the inner loop into the body of the `GenData` procedure. We have to keep in mind that we are inside an ASP page and enclose the procedure header and footer inside `<%` and `%>` braces.

```
<% Sub GenData(x) %>
  <% For y = 1 To 16 %>
    <TD>
      (<%= x %>,<%= y %>)
    </TD>
  <% Next %>
<% End Sub %>
```

The downside of this is that we cannot take this fragment out of the ASP page and reuse it inside an ordinary Visual Basic module. Inside Visual Basic, the only way to produce HTML is by using `Response.Write` to write the HTML as a string.

Hence to *truly* abstract the inner loop, we must first manually apply the page compilation algorithm (in Visual Basic `&` denotes string concatenation), and turn things inside out to write HTML strings to `Response` object. Instead of code embedded inside HTML, we now have unstructured strings that represent HTML embedded inside code:

```
<%
  Sub GenData (x)
    For y = 1 To 16
      Response.Write "<TD bgcolor="
      Response.Write genColor(x,y)
      Response.Write ">"
      Response.Write "(" & x & "," & y & ")"
      Response.Write "</TD>"
    Next
  End Sub
%>
```

The weakness of ASP and almost all other naive server page frameworks based on page translation is that they don't really mix HTML and script; they don't make HTML into a first class citizen. As a result it becomes hard to use the procedural abstraction mechanisms of the scripting language to structure a page. Before we can do so, we first need to translate the server page into its underlying sequence of `Response.Write` statements and then use procedural abstraction to name that sequence of statements.

3 Informal introduction to Haskell Server Pages

The main idea of Haskell Server Pages is to treat HTML (or in fact XML) fragments as ordinary expressions. Inside HTML fragments, strings are not escaped, so we need to escape code fragments instead. For this we reuse the `<%` and `%>` tags. Inside embedded expressions we can of course again introduce HTML fragments, and inside those we can again embed normal expressions by escaping them inside `<%` and `%>`, etc. etc. The meaning of HSP scripts is compositional, and thus easy to understand. Just replace the expression between the `<%` and `%>` by its value.

Attribute values need to be surrounded by double quotes already, hence we can allow any atomic expression to occur in an attribute value position without the need for an escape mechanism.

The following HSP code generates the same 16x16 table as the ASP code we have seen above (the types `TABLE`, `TR` and `TD` are all synonyms for `Element`):

```
table :: TABLE
table =
  <TABLE border="1">
    <% mkRows cells %>
  </TABLE>

cells :: [[(Int,Int)]]
cells =
  [ [ (x,y) | x <- [1..16] ]
  | y <- [1..16]
  ]

mkRows :: [[(Int,Int)]] -> [TR]
mkRows = map $ \cs ->
  <TR><% mkColums cs %></TR>

mkColumns :: [(Int,Int)] -> [TD]
mkColums = map $ \c ->
  <TD bgcolor=(genColor c)>
    <% c %>
  </TD>
```

In this example, and in many of the examples that follow, we use right-associative function application `$` to eliminate parenthesis.

HSP is also implemented using page translation. Instead of translating a page into a sequence of imperative statements, HTML fragments are translated into Haskell expressions using the 'smart' constructor `mkElement :: Name -> Attributes -> [Child] -> Element` and the overloaded `mkChild :: IsChild a => a -> Child` and `mkValue :: IsValue a => a -> Value` functions (a detailed description of the translation is given in section 4):

```
table =
  mkElement "TABLE" [("border", Value "1")] $
    [mkChild (mkRows cells)]

mkRows = map $ \cs ->
  mkElement "TR" [] $
    [mkChild $ mkColums cs]

mkColums = map $ \c ->
  mkElement "TD"
    [("bgcolor", mkValue $ genColor c)] $
      [mkChild c]
```

By using HSP, programmers can write *concrete* HTML in their programs. This shields them from the irrelevant details of using some (arbitrary) encoding of HTML in Haskell, and, most importantly, string literals need not be escaped. At the same time, they have all the power of Haskell's abstraction mechanisms to define and structure functions over such HTML fragments.

3.1 Embedded commands

So far we have only dealt with embedded *expressions*. What about embedded side-effecting *commands*? Note that since in Haskell side-effecting computations are just normal values in the `IO` monad, we need not leave the expression world in order to create them, ie embedded commands are already paid for.

The following example generates an HTML list that records the current date (assuming the existence of the three side-effecting functions `getDay :: IO WeekDay`, `getMonth :: IO Month` and `getYear :: IO Year`):

```
today =
  <UL>
    <LI>Day: <% getDay %> </LI>
    <LI>Month: <% getMonth %> </LI>
    <LI>Year: <% getYear %> </LI>
  </UL>
```

When translated into Haskell, the `mkChild` function leaves embedded commands untouched (in this particular case `mkChild getDay` evaluates to `ChildIO (fmap (Leaf.show) getDay)`).

```
today =
  mkElement "UL" []
    [ ChildElement $ mkElement
        "LI" [] [ Leaf "Day:", mkChild getDay ]
    , ChildElement $ mkElement
        "LI" [] [ Leaf "Day:", mkChild getMonth ]
    , ChildElement $ mkElement
        "LI" [] [ Leaf "Day:", mkChild getYear ]
    ]
```

The function `run :: Element -> IO Element` executes the commands embedded in a document in depth first order.

3.2 Pattern-matching

Being able to construct HTML fragments using concrete syntax is just one side of the coin. Often we also want to do pattern-matching on HTML or XML fragments using concrete syntax.

As an example of using pattern-matching in HSP, we will develop a simple e-mail component that accepts e-mail messages in some XML format and translates them into the standard RFC822 format [12, 19]. Our program will translate the following XML message

```
<MSG>
  <FROM>erik@cs.uu.nl</FROM>
```

```
  <RCPT>
    <TO>billg@microsoft.com</TO>
    <TO>steveb@microsoft.com</TO>
  </RCPT>
  <SUBJECT>HSP</SUBJECT>
  <BODY>
    <P>HSP is cool, isn't it?</P>
    <P>It sure beats ASP!</P>
  </BODY>
</MSG>
```

into the ASCII Internet message

```
From: erik@cs.uu.nl
To: billg@microsoft.com,steveb@microsoft.com
Subject: HSP

HSP is cool, isn't it?
It sure beats ASP!
```

The `toRFC822` function does a match on the top-level `<MSG></MSG>` structure to get to the individual fields of which the message is composed (the types `MSG` and `RFC822` are synonyms for `Element` respectively `RFC822`, but express the intent that their values are valid `MSG` trees respectively valid `RFC822` messages):

```
toRFC822 :: MSG -> RFC822
toRFC822 =
  \<MSG>
    <FROM><% [from] %></FROM>
    <RCPT><% rcpts %></RCPT>
    <SUBJECT><% [subject] %></SUBJECT>
    <BODY><% paras %></BODY>
  </MSG> ->
    concat
      [ "From: " ++ from
      , crlf
      , "To: "
        ++ intersperse "," (stripTOs rcpts)
      , crlf
      , "Subject: " ++ subject
      , crlf
      , crlf
      , intersperse crlf (stripPs paras)
      , crlf
      ]
```

To build the `To` header and the message body, function `toRFC822` uses two functions `stripTOs` and `stripPs` that remove the `<TO></TO>` respectively the `<P></P>` tags:

```
stripTOs :: [TO] -> [String]
stripTOs = map $
  \<TO><% [to] %></TO> -> to

stripPs :: [P] -> [String]
stripPs = map $
  \<P><% [p] %></P> -> p
```

HSP will match a single `<%[`p`]%>`child to the pattern `[Leaf` p`]` of a singleton list of a `Leaf` child, and a single `<%`p`%>` child to p itself. All other child patterns are translated into a finite list. Hence the pattern that appears in the `toRFC822`

```
<MSG>
  <FROM><% [from] %></FROM>
  <RCPT><% rcpts %></RCPT>
  <SUBJECT><% [subject] %></SUBJECT>
  <BODY><% paras %></BODY>
</MSG>
```

is translated by the HSP preprocessor into the following normal Haskell pattern

```
Node "MSG" []
  [ ChildElement (Node "FROM"    [] [Leaf from])
  , ChildElement (Node "RCPT"    [] rcpts)
  , ChildElement (Node "SUBJECT" [] [Leaf subject])
  , ChildElement (Node "BODY"    [] paras)
  ]
```

This convention turns might perhaps look odd at first sight, but in practice it turns out to be very convenient to circumvent the inability to express the regular expression matching that is needed to do 'real' pattern-matching on XML documents.

4 Embedding and Translation

Now that we have seen a number of examples of using HSP, it is time to get down to the nitty gritty details of the HSP implementation.

4.1 Syntax

Syntactically, HSP simply adds XML fragments as possible atomic Haskell expressions and patterns.

$$
\begin{array}{rcl}
aexp & \rightarrow & qvar \\
 & | & gcon \\
 & | & literal \\
 & | & (exp) \\
 & | & \ldots \\
 & | & xml
\end{array}
$$

$$
\begin{array}{rcl}
xml & \rightarrow & \texttt{<}name\ attrs\texttt{>}\ child_1 \ldots child_n\ \texttt{</}name\texttt{>} \\
 & | & \texttt{<}name\ attrs\texttt{/>}
\end{array}
$$

Inside XML fragments, strings (*PCDATA* in XML lingo) need not be enclosed inside double quotes, but Haskell code needs to be escaped using `<%` and `%>` brackets.

$$
\begin{array}{rcl}
child & \rightarrow & PCDATA \\
 & | & xml \\
 & | & \texttt{<\%}\ expr\ \texttt{\%>}
\end{array}
$$

Attributes are simple *name* `=` *value* pairs, where the value can be any atomic Haskell expression.

$$
attrs \rightarrow name \texttt{ = } aexp \ldots name \texttt{ = } aexp
$$

As usual, pattern syntax is a close copy of the expression syntax. In our case we extend Haskell's *apat* production with a new alternative for XML patterns *xpat*. The productions for the nonterminal *xpat* are the same as those for *xml* but replacing *exp* everywhere by *pat* and *aexp* by *apat*.

4.2 Semantics

4.2.1 Representation

Our current implementation assumes a universal document representation for XML fragments that closely follows the concrete grammar given above:

```
type Name = String

data Element
  = Node Name Attributes [Child]

data Child
  = Leaf String
  | ChildElement Element
  | ChildList [Child]
  | ChildIO (IO Child)
```

The only surprises are perhaps the `ChildList` and `ChildIO` constructors. The `ChildIO` constructor encapsulates an embedded command that, when executed, will produce a child document. The `ChildList` constructor is necessary because embedded child-expressions might return a list of child elements themselves, as in the following simple example:

```
<P> the <% ["quick", "brown"] %> fox </P>
```

In the translation from HSP to plain Haskell we will use a 'smart' constructor for `Element` that flattens its children to remove `ChildList`.

Since HSP attributes can be any Haskell expression, we also introduce a `ValueIO` constructor to encapsulate an embedded command, that when executed will return a `Value`.

```
type Attributes = [(Name,Value)]

data Value
  = Value String
  | ValueIO (IO Value)
```

Note that nothing prevents us from using one of the more strongly typed representations that have been proposed for representing HTML or XML documents [34, 8, 33] in Haskell. As we have argued before, the Haskell type system is fundamentally too weak to truly embed the XML type system. We have chosen the simplest possible solution over more complicated, but nonetheless partial ones.

4.2.2 Translation

The semantics of HSP is defined by a handful of simple translation functions that map HSP scripts into pure Haskell. The types of the translation functions show the source and target grammar productions. For the target, we also indicate the required type of the resulting phrase.

Function $\mathcal{X}\llbracket\ \rrbracket$ translates XML fragments into applications of the `mkElement` function on the recursive translations of the attributes and children:

$$
\begin{aligned}
&\mathcal{X}\llbracket\ \rrbracket \in \mathit{xml} \rightarrow \mathit{aexp} :: \texttt{Element}\\
&\mathcal{X}\llbracket\texttt{<}e\ \mathit{as}\texttt{>}c_1 \ldots c_n\texttt{</}e\texttt{>}\rrbracket\\
&\quad = \ \texttt{(mkElement "}e\texttt{"}\ \mathcal{A}\llbracket \mathit{as}\rrbracket\ \texttt{[}\mathcal{C}\llbracket c_1\rrbracket\texttt{,}\ldots\texttt{,}\mathcal{C}\llbracket c_n\rrbracket\texttt{])}
\end{aligned}
$$

Function `mkElement` is the 'smart' constructor we promised above that flattens its children to a list of `Element`, `Leaf`, and `ChildIO` children by calling the function `flatten :: [Child] -> [Child]`:

```
mkElement :: Name -> Attributes -> [Children] ->
  Element
mkElement = \tag -> \attributes -> \children
  Node tag attributes (flatten children)
```

Function $\mathcal{C}\llbracket\ \rrbracket$ translates a *PCDATA* child into a corresponding Haskell string:

$$
\begin{aligned}
\mathcal{C}\llbracket\ \rrbracket &\in \mathit{PCDATA} \rightarrow \mathit{expr} :: \texttt{Child}\\
\mathcal{C}\llbracket s\rrbracket &= \texttt{Leaf "}s\texttt{"}
\end{aligned}
$$

If we were really pedantic we should also define how XML entities (such as `>`) that can occur in *PCDATA* strings are translated into Haskell.

A normal XML child is translated recursively and wrapped as an `ChildElement`:

$$
\begin{aligned}
\mathcal{C}\llbracket\ \rrbracket &\in \mathit{xml} \rightarrow \mathit{expr} :: \texttt{Child}\\
\mathcal{C}\llbracket x\rrbracket &= \texttt{ChildElement}\ \mathcal{X}\llbracket x\rrbracket
\end{aligned}
$$

An embedded Haskell expression wraps a `mkChild` around the translation of the expression so that it has the required `Child` type:

$$
\begin{aligned}
\mathcal{C}\llbracket\ \rrbracket &\in \texttt{<\%}\mathit{expr}\texttt{\%>} \rightarrow \mathit{expr} :: \texttt{Child}\\
\mathcal{C}\llbracket\texttt{<\%}\ e\ \texttt{\%>}\rrbracket &= \texttt{mkChild}\ \mathcal{E}\llbracket e\rrbracket
\end{aligned}
$$

The $\mathcal{E}\llbracket\ \rrbracket$ translation function removes all nested XML fragments in an embedded Haskell expression:

$$
\begin{aligned}
\mathcal{E}\llbracket\ \rrbracket &\in \mathit{expr} \rightarrow \mathit{expr}\\
\mathcal{E}\llbracket e[x]\rrbracket &= e[\mathcal{X}\llbracket x\rrbracket]
\end{aligned}
$$

The overloaded function `mkChild` of the `IsChild` type class transforms values into children of XML elements.

```
class IsChild a where
  { mkChild :: a -> Child }
```

Assuming overlapping instances, we provide standard instances for any type that is an instance of `Show` in conjunction with those for `Char`, `String`, `[]` and `IO`:

```
instance Show a => IsChild a where
  { mkChild a = Leaf (show a) }

instance IsChild Char where
  { mkChild c = (Leaf . dropQuotes . show) c }

instance IsChild String where
  { mkChild s = (Leaf . dropQuotes . show) s }

instance IsChild a => IsChild [a] where
  { mkChild as = ChildList (map mkChild as) }

instance IsChild a => IsChild (IO a) where
  { mkChild ma = ChildIO (fmap mkChild ma) }
```

The translation function $\mathcal{A}\llbracket\ \rrbracket$ for attributes translates each value using $\mathcal{V}\llbracket\ \rrbracket$.

$$\mathcal{A}\llbracket\ \rrbracket \in \mathit{attrs} \to \mathit{aexpr} :: \texttt{Attributes}$$
$$\mathcal{A}\llbracket n_1 = e_1\ \ldots\ n_n = e_n \rrbracket$$
$$=\quad \texttt{[("}n_1\texttt{",}\mathcal{V}\llbracket e_1\rrbracket\texttt{)}\ \ldots\ \texttt{("}n_n\texttt{",}\mathcal{V}\llbracket e_n\rrbracket\texttt{)]}$$

The translation function for attribute values $\mathcal{V}\llbracket\ \rrbracket$ inserts the overloaded function `mkValue` when the value is not a string literal.

$$\begin{array}{rcl}
\mathcal{V}\llbracket\ \rrbracket & \in & \mathit{string} \to \mathit{aexp} :: \texttt{Value} \\
\mathcal{V}\llbracket \texttt{"}s\texttt{"} \rrbracket & = & \texttt{Value "}s\texttt{"} \\
\mathcal{V}\llbracket\ \rrbracket & \in & \mathit{aexp} \to \mathit{aexp} :: \texttt{Value} \\
\mathcal{V}\llbracket e \rrbracket & = & \texttt{mkValue}\ e
\end{array}$$

The instances of class `IsValue` are similar to those of the `IsChild` class, so we won't repeat them here.

```
class IsValue a where
  { mkValue :: a -> Value }
```

Having seen the translation of expressions, the translation of patterns should pose no problems. The only thing to note is the special cases when an element has exactly one embedded Haskell child.

$$\mathcal{XP}\llbracket\ \rrbracket \in \mathit{xpat} \to \mathit{apat} :: \texttt{Element}$$
$$\mathcal{XP}\llbracket \texttt{<}e\ \mathit{as}\texttt{><\%}\ [p]\ \texttt{\%></}e\texttt{>} \rrbracket$$
$$= \ \texttt{Element "}e\texttt{"}\ \mathcal{AP}\llbracket as \rrbracket\ \texttt{[Leaf}\ \mathcal{P}\llbracket p \rrbracket\texttt{]}$$
$$\mathcal{XP}\llbracket \texttt{<}e\ \mathit{as}\texttt{><\%}\ p\ \texttt{\%></}e\texttt{>} \rrbracket$$
$$= \ \texttt{Element "}e\texttt{"}\ \mathcal{AP}\llbracket as \rrbracket\ \mathcal{P}\llbracket p \rrbracket$$
$$\mathcal{XP}\llbracket \texttt{<}e\ \mathit{as}\texttt{>}c_1 \ldots c_n\texttt{</}e\texttt{>} \rrbracket$$
$$= \ \texttt{Element "}e\texttt{"}\ \mathcal{AP}\llbracket as \rrbracket\texttt{[}\mathcal{CP}\llbracket c_1 \rrbracket\texttt{,} \ldots \texttt{,}\mathcal{CP}\llbracket c_n \rrbracket\texttt{]}$$

Patterns occurring in child position are straightforward as well. Leaf and element children introduce their respective constructors:

$$\begin{array}{rcl}
\mathcal{CP}\llbracket\ \rrbracket & \in & \mathit{PCDATA} \to \mathit{pat} :: \texttt{Child} \\
\mathcal{CP}\llbracket s \rrbracket & = & \texttt{Leaf "}s\texttt{"} \\
\mathcal{CP}\llbracket\ \rrbracket & \in & \mathit{xpat} \to \mathit{pat} :: \texttt{Child} \\
\mathcal{CP}\llbracket x \rrbracket & = & \texttt{ChildElement}\ \mathcal{XP}\llbracket x \rrbracket
\end{array}$$

Embedded Haskell patterns are recursively translated using $\mathcal{P}\llbracket\ \rrbracket$:

$$\begin{array}{rcl}
\mathcal{CP}\llbracket\ \rrbracket & \in & \texttt{<\%}\ \mathit{pat}\ \texttt{\%>} \to \mathit{pat} :: \texttt{Child} \\
\mathcal{CP}\llbracket \texttt{<\%}\ p\ \texttt{\%>} \rrbracket & = & \mathcal{P}\llbracket p \rrbracket \\
\mathcal{P}\llbracket\ \rrbracket & \in & \mathit{pat} \to \mathit{pat} \\
\mathcal{P}\llbracket p[x] \rrbracket & = & p[\mathcal{XP}\llbracket x \rrbracket]
\end{array}$$

The translation of attribute patterns uses an auxiliary translation function $\mathcal{VP}\llbracket\ \rrbracket$ for values:

$$\mathcal{AP}\llbracket\ \rrbracket \in \mathit{pattrs} \to \mathit{apat} :: \texttt{Attributes}$$
$$\mathcal{AP}\llbracket n_1 = p_1\ \ldots\ n_n = p_n \rrbracket$$
$$=\quad \texttt{("}n_1\texttt{",}\mathcal{VP}\llbracket p_1\rrbracket\texttt{):}\ \ldots\ \texttt{:("}n_n\texttt{",}\mathcal{VP}\llbracket p_n\rrbracket\texttt{):}\ \_$$

Function $\mathcal{VP}\llbracket\ \rrbracket$ takes care of the special case of a literal string pattern occurring as an attribute value.

$$\begin{array}{rcl}
\mathcal{VP}\llbracket\ \rrbracket & \in & \mathit{string} \to \mathit{apat} :: \texttt{Value} \\
\mathcal{VP}\llbracket \texttt{"}s\texttt{"} \rrbracket & = & \texttt{Value "}s\texttt{"} \\
\mathcal{VP}\llbracket\ \rrbracket & \in & \mathit{apat} \to \mathit{apat} :: \texttt{Value} \\
\mathcal{VP}\llbracket p \rrbracket & = & p
\end{array}$$

5 The HSP runtime

To use HSP to write dynamic web pages, we need to provide interaction with a web server and the middle tier runtime environment. To make HSP really useful we must not tie ourselves too close to a particular web server. The HSP runtime provides a number of components (`Request`, `Response`, `Application`, and `Session`) that assist in handling HTTP requests and responses, and to maintain (session and application) state across different invocations of a HSP application (a set of HSP scripts). Our choice is influenced by the ASP built-in objects, but similar objects are available in one form or another in most other server-side scripting frameworks (Java Servlets, CGI, Apache) [5].

We pass the HSP runtime components as implicit arguments [18] to scripts. In this way we do not need to explicitly pass object references, but nevertheless they show up in the type of the functions that use them. We don't provide public access to the real objects, so we can enforce constraints on the use of these objects. The programmer simply cannot bind the implicit arguments to a value (except for `undefined`, but that is the price we have to pay for having full recursion and partial functions).

In section 5.1, we describe several events that the HSP runtime will fire and how to handle these in HSP. For instance when the first page in an application is about to be served, the HSP runtime will first call the event handler `onApplicationStart :: (?a :: Application) => IO ()`. When this event occurs, only the `Application` object is available, which is precisely captured in the handler's type.

Request The `Request` object contains the information passed from the client to the server through an HTTP request. Function `getParameter` gives access to the query string, while `getServerVariable` gives access to the environment variables set by the server for this request.

```
getParameter ::
  (?q :: Request, Read a) => String -> [a]
getServerVariable ::
  (?q :: Request) => String -> String
```

For example we can make the call `getServerVariable "HTTP_USER_AGENT"` to learn which browser has sent the request. And if the client request was `.../example.hsp?language=haskell&version=98` then in the script `example.hsp` we could ask `getParameter "version"` to read the value of the `version` parameter in the query string.

Application The `Application` object stores information that is accessible by the application as a whole, ie that is shared by all sessions. Access to the `Application` object is automatically synchronized.

```
getApplication ::
  (?a :: Application, Read a) =>
    String -> IO a
setApplication ::
  (?a :: Application, Show a) =>
    String -> a -> IO ()
```

A typical use of the application object is to count the total number of times a page has been visited (see the example in section 5.2).

Session The `Session` object stores information for each individual user of the application. A `Session` object is created whenever a new user enters the application, and is destroyed when the server terminates the whole application, the session times out (in a web application you can never know when or wether the user will return), or is explicitly abandoned by the script.

The functions `getSession` and `setSession` implement variables with session-wide scope, and life time. A typical use for the `Session` object is to store user preferences (eg with/without frames) or shopping carts in e-commerce applications.

```
getSession ::
  (?s :: Session, Read a) => String -> IO a
setSession ::
  (?s :: Session, Show a) =>
    String -> a -> IO ()
```

The `sessionID` function returns the unique ID that the server has generated for this session. The server usually stores this information in a cookie, so things will break if the client does not accept cookies.

```
sessionID :: (?s :: Session) => String
```

If the user does not request another page within `TimeOut` minutes, the session is terminated by the server. Scripts can explicitly ask to end the current session (after the script has terminated of course) by calling the `abandon` method.

```
setTimeOut :: (?s :: Session) => Int -> IO ()
getTimeOut :: (?s :: Session) => IO Int
abandon :: (?s :: Session) => IO ()
```

5.1 HSP Application

A HSP application consists of a virtual directory and all its subdirectories. The root-directory should contain a file `Global.hsp` that contains definitions for four callback functions.

The `onApplicationStart` function is called before the first new session is created, ie before the first call to `onSessionStart`. At that moment, only the `Application` object is available. The type of `onApplicationStart` enforces this constraint.

```
onApplicationStart :: (?a :: Application) => IO ()
```

When the application quits, the HSP runtime invokes the `onApplicationEnd` (after the last call to `onSessionEnd`). At that moment, only the `Application` object is available; again, the type of `onApplicationEnd` enforces this constraint.

```
onApplicationEnd :: (?a :: Application) => IO ()
```

When the server creates a new session, it first invokes the `onSessionStart` function before serving the page. As the type of `onSessionStart` shows, all the built-in objects are available at that time.

```
onSessionStart ::
  ( ?a :: Application
  , ?s :: Session
  , ?q :: Request
  ) => IO ()
```

When a session is abandoned or times out, the runtime calls `onSessionEnd` function. As the type shows, only the `Application` and `Session` objects are available at that time.

```
onSessionEnd ::
  ( ?a :: Application
  , ?s :: Session
  ) => IO ()
```

5.2 A complete HSP application

As an example of a complete HSP application, we will program a page that maintains two page counters. One counter counts the total number of visitors that have accessed the page since the application was started, the other counter counts the number of times an individual user of the application has accessed the page.

The two counters are initialized in the `onApplicationStart` and `onSessionStart` functions in the `Global.hsp` file.

```
module Global
  ( onSessionStart
  , onSessionEnd
  , onApplicationStart
  , onApplicationEnd
  )
where

onSessionStart =
  do{ setSession "Count" 0 }

onApplicationStart =
  do{ setApplication "Count" 0 }
```

The functions `onApplicationEnd` and `onSessionEnd` don't do anything interesting so we have omitted their definitions.

The page uses two helper functions `countPage` and `countSession`, that are in turn defined using a helper function `count`.

```
count = \getter -> \setter
  do{ n <- getter
    ; let n' = n+1
    ; setter n'
    ; return n'
    }

countPage :: (?a :: Application) -> IO Int
countPage = count
  (getApplication "Count") (setApplication "Count")

countSession :: (?s :: Session) -> IO Int
countSession = count
  (getSession "Count") (setSession "Count")
```

The code for the page itself uses a small embedded script to display the page counter as a sequence of gifs for each digit.

```
page ::
  ( ?a :: Application
  , ?s :: Session
  ) => Element
page =
  <HTML>
    <TITLE>Counter</TITLE>
  <BODY>
    <P>
      This page has been accessed
      <%
        do{ n  <- countPage
          ; return
              [ <IMG SRC=(i:".gif") />
              | i <- show n
              ]
          }
      %>
      times.
    </P>
    <P>
      You have visited this page
      <% countSession %> times.
    </P>
  </BODY>
  </HTML>
```

5.3 Implementation of the HSP runtime

We have designed the HSP runtime such that it can be implemented using the services that most middle tier frameworks and web servers offer. Here we will sketch an implementation on top of COM+ and Active Server Pages; the architecture of solutions that use Apache and C is the same, and only differ in the implementation details.

Each HSP page *MyPage*`.hsp` should export a function

```
page ::
  ( ?a :: Application
  , ?s :: Session
  , ?q :: Request
  ) => Element
```

that will generate the response when that page is requested. The HSP page is compiled into a normal Haskell file *MyPage*`.hs`. The HSP compiler also generates a wrapper module *MyPage*`Wrapper.hs` and an ASP page *MyPage*`.asp`.

The task of the wrapper module *MyPage*`Wrapper.hs` is to pass the implicit server objects to the `page` function in *MyPage*`.hs`.

```
module MyPageWrapper (page) where

import ServerObjects
import qualified MyPage

page ::
  Application -> Session ->
  Request -> Response -> IO ()
page = \a -> \s -> \q -> \r ->
  in do{ p <- run $ MyPage.page
                with {?a = a; ?s = s; ?q = q }
       ; r # write (show p)
       }
```

The ASP page *MyPage*`.asp` is the page that the server will execute when *MyPage*`.hsp` is requested. By calling the `GeneratePage` method, the ASP page asks the `HSP` component to generate the page using the `page` function in the Haskell module *MyPage*.

```
<%
  HSP.GeneratePage "MyPage" _
    Application, Session, Request, Response
%>
```

The `HSP` object is created in the `global.asa` file of the HSP application

```
<OBJECT
   RUNAT=Server SCOPE=Application
   ID=HSP PROGID="HSP.Server">
</OBJECT>
```

The `HSP.Server` component is implemented in Visual Basic. The `GeneratePage` method will immediately delegate the call to corresponding `page` function in Haskell using the DietHEP component (see below):

```
Sub GeneratePage _
  ( ByVal m As String _
  , ByVal a As Application _
  , ByVal s As Session _
  , ByVal q As Request _
  , ByVal r As Response _
  )
  Long h = DietHEP.LoadLibrary (m & Wrapper)
  Addr p = DietHEP.GetProcAddress (stdcall,m,"page")
  Page p, a, s, q, r
End Sub
```

In Visual Basic it is possible to call function pointers[3] using a helper function `CallAddr` that is exported from the `DynamicImport` DLL [2]. To call the Haskell `page` function via its address, we declare a Visual Basic subroutine `Page`:

```
Declare Sub Page _
  Lib "DynamicImport.dll" Alias "CallAddr" _
    ( ByVal Addr As Long _
    , ByVal a As Application _
    , ByVal s As Session _
    , ByVal q As Request _
    , ByVal r As Response _
    )
```

This implementation of the HSP runtime ultimately relies on the fact that Haskell functions can be exported as ordinary "C" function pointers. The DietHEP component [30] leverages on the foreign function interface we defined for Haskell in previous papers [7] and allows us to view *any* Haskell module as an ordinary DLL. Clients of DietHEP see no difference (except for the extra flexibility of specifying the calling convention at runtime) between using an ordinary DLL via the `kernel32.dll` or using the `DietHEP` primitives.

The `LoadLibrary` function takes the name of a Haskell module (or GHC compiled binary), loads it and returns a *handle* to the module. Function `GetProcAddress` takes that handle and a function name and returns a 'foreign export'-ed version of the requested Haskell function.

[3]Note that this is rather similar to Haskell's `foreign import dynamic` mechanism.

```
[dllname("DietHEP.dll")]
module DietHEP {
   typedef enum { stdcall, ccall } CALLCONV;

   HMODULE LoadLibrary
     ( [in,string]char* modname );
   FARPROC  GetProcAddress
     ( [in]CALLCONV cconv
     , [in]HMODULE hModule
     , [in,string]char* lpProcName
     );
};
```

To summarize, a HSP page is served by a shadow ASP page, that calls a method on an object implemented in Visual Basic, that will call the wrapper function in Haskell, that will finally call the function that generates the page itself.

6 Implementation status

At the time of writing this paper, we have a prototype version of HSP that runs under Apache and `mod_Haskell` on Linux. The current HSP preprocessor does not yet use a full Haskell parser but parses `.hsp` documents as XML. Hence, it cannot distinguish between patterns and expressions and does not know about the layout rule. To compensate for this dumbness, the programmer is required to use special tags `<#` and `#>` for patterns and use explicit semicolons and braces in embedded code. Once DietHEP is stable and runs under Windows, we will provide the fully integrated version of HSP as defined in this paper. The current prototype is available for download at `http://www.microsoft-lab.org/HSP`.

Acknowledgements

Many thanks to Armijn Hemel, Eelco Dolstra, and the other students of the *Internet Programming* course for their work in implementing the prototype version of HSP and integrating it into `mod_Haskell` and IIS. Thanks also to the Haskell Workshop referees, Nigel Perry, Gert Florijn, Arjan van IJzendoorn, Frank Atanassow, Koen Claessen, and especially to Daan Leijen and Simon Peyton Jones for their comments on various drafts of this paper.

References

[1] Kirby W. Angell. Python Server Pages (PSP), Part I. *Dr Dobbs Journal*, January 2000.

[2] Microsoft Cooperation. HOWTO: Do Generic Callbacks Using a Helper DLL. http://support.microsoft.com/support/kb/articles/-Q171/7/29.asp.

[3] Microsoft Cooperation. Microsoft Windows Script Interfaces-Introduction. Platform SDK.

[4] David Flanagan. *JavaScript Pocket Reference.* O'Reilly & Associates, 1998.

[5] H.M. Deitel, P.J. Deitel, and T.R. Nieto. *Internet & World Wide Web –How to Program–.* Prentice-Hall, 2000.

[6] Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1, 1 October 1998.

[7] Sigbjorn Finne, Erik Meijer, Daan Leijen, and Simon Peyton Jones. HDirect: A Binary Foreign Function Interface for Haskell. In *Proceedings of ICFP'98.*

[8] Andy Gill. HTML combinators (version 0.2). http://www.cse.ogi.edu/~andy/html/index.htm.

[9] Scot Hillier and Daniel Mezick. *Programming Active Server Pages.* Microsoft Press, 1997.

[10] Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000.

[11] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[12] Lawrence Hughes. *Internet e-mail: Protocols, Standards, and Implementation.* Artech House Publishers, 1998.

[13] Johan Vromans. *Perl 5 Pocket Reference.* O'Reilly & Associates, 1998.

[14] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM Components in Haskell. In *Proc 5th International Conference on Software Reuse*, 1998.

[15] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. In *2nd USENIX Conference on Domain-Specific Languages*, 1999.

[16] Daan Leijen, Erik Meijer, and Jim Hook. *Haskell as an Automation Controller.* LNCS 1608. 1999.

[17] Rasmus Lerdorf. *PHP Pocket Reference.* O'Reilly & Associates, 2000.

[18] Jeff Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit Arguments: Dynamic Scoping with Static Types. In *Proceedings of POPL'00.*

[19] Pete Loshin. *Essential e-mail Standards: RFCs and Protocols Made Practical.* John Wiley & Sons, Inc., 2000.

[20] Stefano Mazzocchi and Ricardo Rocha. eXtensible Server Pages (XSP). http://java.apache.org/-cocoon/xsp/WD-xsp.html, Working Draft2000-01-09.

[21] Erik Meijer. Server-side Scripting in Haskell. *Journal of Functional Programming*, 2000.

[22] Erik Meijer and Sigbjorn Finne. Lambada, Haskell as a better Java. In these proceedings, 2000.

[23] Erik Meijer, Daan Leijen, and Jim Hook. Client-side Web Scripting with HaskellScript. In *PADL*, 1999.

[24] Erik Meijer and Mark Shields. XMLambda: A Functional Programming Language for Constructing and Manipulating XML Documents. http://www.cse.ogi.edu/~mbs, 2000.

[25] Erik Meijer, Arjan van IJzendoorn, and Nigel Perry. Report on the Language Mondrian. htpp://www.mondrian-script.org/report, August 2000.

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

[27] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, 1986.

[28] Scot Hillier. *Inside Microsoft Visual Basic Scripting Edition*. Microsoft Press, 1996.

[29] Roger Sessions. *COM+ and the Battle for the Middle Tier*. John Wiley & Sons, 2000.

[30] Julian Seward, Simon Marlow, Andy Gill, Sigbjorn Finne, and Simon Peyton Jones. Architecture of the Haskell Execution Platform (HEP) Version 6. http://www.haskell.org/ghc/docs/papers/hep.ps.gz, July 1999.

[31] Mark Shields and Erik Meijer. Type Indexed Rows. Submitted for publication.

[32] R.D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

[33] Peter Thiemann. Modeling HTML in Haskell. In *PADL'00*, LNCS 1753.

[34] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *ICFP*, 1999.

[35] Extensible Stylesheet Language (XSL) Specification. http://www.w3.org/TR/WD-xsl, 21 April 1999.