sestoft@dina.kvl.dk

ML Server Pages

- ML Server Pages (MSP) is a loose integration of SML and HTML
- MSP is modelled on Active Server Pages (ASP), Java Server Pages (JSP), and PHP3
- We describe a proof-of-concept implementation using Moscow ML and the Apache webserver

Plan

- MSP examples
- MSP notation
- Implementation: MSP and the Apache webserver
- Practice: Efficient functional generation of HTML code

For MSP description and examples, see

`http://ellemose.dina.kvl.dk/˜sestoft/msp/index.msp`

MSP example script: Hello world

An MSP script contains a mixture of HTML fragments and ML fragments, enclosed in `<% ... %>`.

An MSP script is edited and stored as a plain HTML file in your usual HTML directory.

Example: File `hello.msp`:

```
<HTML><BODY>

<H1>Hello world!</H1>

The current date and time is
<%= Date.toString (Date.fromTimeLocal(Time.now())) %>

<HR><ADDRESS>Your friendly ML server page</ADDRESS>
</BODY></HTML>
```

When requested from the webserver, the above MSP script generates the following HTML code:

```
<HTML><BODY>

<H1>Hello world!</H1>

The current date and time is
Sat Jan 22 21:23:43 2000

<HR><ADDRESS>Your friendly ML server page</ADDRESS>
</BODY></HTML>
```

MSP notation

ML fragments are written within the pseudo-tags `<% ... %>`.

An ML fragment may declare SML types and values, and may print or generate HTML code which is embedded in the resulting HTML page.

There are four kinds of ML fragments:

- `<% dec %>`

  The SML declaration `dec` is executed to define functions and variables, and to print HTML code.
- `<%= exp %>`

  The SML expression `exp` must have type `string`. It is evaluated and its result is printed.
- `<%! exp %>`

  The SML expression `exp` must have type `unit`. It is evaluated for its side effects only. Typically, it prints HTML code.
- `<%$ exp %>`

  The SML expression `exp` must have type `Msp.wseq`, a representation of efficiently concatenable strings. It is evaluated and the resulting wseq is printed.

MSP example script: A database query

```
<HTML><HEAD><TITLE>MSP example: database query</TITLE></HEAD><BODY>

<H2>MSP example: database queries</H2>

<% (* Open database connection *)
val db = Postgres.openbase
         { dbhost    = NONE, dbname    = SOME "messages", dboptions = NONE,
           dbport    = NONE, dbpwd     = NONE,            dbtty     = NONE,
           dbuser    = SOME "nobody" };
%>

<H2>Example database query</H2>
We open a database connection db and execute the following SML code
<PRE>
    Msp.pgshowquery db
    "SELECT * FROM message WHERE name = 'Peter Sestoft' ORDER BY day"
</PRE>

<P>The result is this table, automatically generated by Msp.pgshowquery:
<P>
<%$ Msp.pgshowquery db
    "SELECT * FROM message WHERE name = 'Peter Sestoft' ORDER BY day"
%>
```

The SML variable `db` from the first ML (declaration) fragment is visible in the second ML (expression) fragment.

Implementation: MSP under the Apache webserver

When receiving a request for an MSP script, the webserver must execute an ML program derived from the script.

Thus we configure the Apache webserver so that

- it maps the file extension `.msp` to a new MIME type called `application/x-msp`;
- it invokes the CGI script `/cgi-bin/mspcompile` to handle the MIME type `application/x-msp`

This is achieved by adding just two lines to an Apache configuration file:

```
Action application/x-msp /cgi-bin/mspcompile
AddType application/x-msp .msp
```

The `/cgi-bin/mspcompile` script does the following:

- it checks that a compiled version of the requested `.msp` file exists and is up to date;
- if that isn't true, then it generates an SML CGI script from the `.msp` file and compiles it;
- if compilation fails, it reports an error back to the client (as an HTML page);
- in all other cases, it invokes the compiled script to generate HTML code, which is sent to the client.

Implementation: How `mspcompile` generates an SML CGI script

The `.msp` file must be split into HTML fragments and ML fragments (those enclosed in `<% ... %>`).

ML fragments are recognized by this regular expression (using Moscow ML structure Regex):

```
val mlfrag = Regex.regcomp "<%([^%]*(%([^>][^%]*)?)*)%>" [Regex.Extended];
```

We transform the HTML and ML fragments while recognizing them:

```
Regex.fold mlfrag (mkstringval out, mkmlfrag outsus) () mspsrc;
```

Function mkstringval transforms HTML fragments, while preserving the formatting of the HTML source (e.g., for `<PRE>`), and generating efficient and readable SML code. For instance, the HTML fragment

```
<PRE>
    Msp.pgshowquery db
    "SELECT * FROM message WHERE name = ’Peter Sestoft’ ORDER BY day"
</PRE>
```

is transformed into the SML code

```
val _ = print
"<PRE>\n\
\    Msp.pgshowquery db \n\
\    \"SELECT * FROM message WHERE name = ’Peter Sestoft’ ORDER BY day\"\n\
\</PRE>\n";
```

This is done by the SML Basis Library functions Substring.fields, Substring.translate, Char.toString (for ‘lifting’).

Function mkmlfrag inserts an ML fragment into the SML code as is, wrapped in `val _ = (...);` or similar.

The SML code generated for the Hello world script

```
val _ = print "Content-type: text/html\n\n"
val _ = print "<HTML><BODY>\n\
\\n\
\<H1>Hello world!</H1>\n\
\\n\
\The current date and time is\n\
\";
val _ = print( Date.toString (Date.fromTimeLocal(Time.now())) );
val _ = print "\n\
\\n\
\<HR><ADDRESS>Your friendly ML server page</ADDRESS>\n\
\</BODY></HTML>\n\
\";
```

Practice: Efficient functional HTML code generation

The ML fragments of an MSP script often generate highly structured HTML code: nested tables, forms, etc.

The most general and transparent approach is to do this functionally, without side effects.

However, repeated concatenation of SML strings requires copying and should be avoided:

```
"<HTML><BODY>" ^ docbody ^ "</BODY></HTML>"
```

Instead we use efficiently concatenable word sequences (type `Msp.wseq`):

```
datatype wseq =
    Empty                               (* The empty sequence         *)
  | Nl                                  (* Newline                    *)
  | $ of string                         (* A string                   *)
  | $$ of string list                   (* A sequence of strings      *)
  | && of wseq * wseq;                  (* Concatenation of sequences *)
infix &&
```

Now we can write (and execute in constant time):

```
$"<HTML><BODY>" && docbody && $"</BODY></HTML>"
```

Why not define an SML datatype for representing HTML directly?

Because (1) HTML is too vaguely defined in practice, and (2) we want to handle non-HTML text similarly.

Practice: HTML tags as `wseq` functions

The structure Msp defines wseq-generating functions for the most common HTML tags, and some utilities:

```
val html  : wseq -> wseq
val head  : wseq -> wseq
val title : wseq -> wseq
val body  : wseq -> wseq
val bodya : string -> wseq -> wseq
...
val prmap : ('a -> wseq) -> 'a list -> wseq
val prsep : wseq -> ('a -> wseq) -> 'a list -> wseq
```

The application `body ws` generates `<BODY>ws</BODY>` etc.

The a-versions `bodya attr ws` generate `<BODY attr>ws</BODY>`, that is, tags with attributes.

Now we may write the example less obscurely:

```
html (body docbody)
```

Some higher-order wseq functions:

`prmap f [x1, ..., xn]` is equivalent to `f x1 && ... && f xn`

`prsep sep f [x1, ..., xn]` is equivalent to `f x1 && sep && ... && sep && f xn`

For instance, formatting a list `elts` of numbers as an HTML table row:

```
tr (prmap (td o $ o Int.toString) elts)
```

Practice: Size and efficiency

The `mspcompile` script is 150 lines of SML. The size of the compiled script (incl. runtime) is 96 KB.

The overhead for invoking the `mspcompile` script is less than 0.01 second.

Generating, compiling, and linking `database.msp` (80 lines MSP) takes 0.20 sec.

Most of the time is spent compiling and linking the generated SML code (84 lines).

The compiled and linked `database` script is 21 KB.

The compiled script generates 19 KB HTML in 0.35 sec (0.10 sec script plus 0.25 sec database server).

Times are for a 266 MHz Pentium II notebook running SuSE Linux 6.1, Postgres 6.3, and Moscow ML 1.44.

Example: Formatting the result of a database query

Function Msp.pgformattable takes a database query result `dbres` and returns an HTML table:

```
fun pgformattable (dbres : Postgres.dbresult) =
    let open Postgres
        fun (f o g) x = f (g x)
        val fieldnms = prmap (th o $) (vec2list (fnames dbres))
        fun fmtval dynval =
            case dynval of
                Int  _ => tda "ALIGN=RIGHT" ($ (dynval2s dynval))
              | Real _ => tda "ALIGN=RIGHT" ($ (dynval2s dynval))
              | _      => td ($ (dynval2s dynval))
        fun fmtrow tuple = tr(prmap fmtval (vec2list tuple))
        val tuples = vec2list (getdyntups dbres)
    in
        tablea "BORDER" (tr fieldnms && prsep Nl fmtrow tuples)
    end
```

The number of columns and rows in the HTML table depends on the schema and tuples of the relation.

The HTML table has a header with the relation's field names.

Number fields are right-justified; all other data are left-justified.

The function uses HTML generators prmap, prsep, tablea, tr, th, td, and tda from structure Msp, and database access functions fnames, dynval2s, and getdyntups from structure Postgres.