MetaOCaml Server Pages: Web Publishing as Staged Computation

Christopher League

Long Island University · Computer Science
1 University Plaza · Brooklyn, NY 11201
christopher.league@liu.edu

Abstract

Modern dynamic web services are really computer programs. Some parts of these programs run off-line, others run server-side on each request, and still others run within the browser. In other words, web publishing is *staged computation,* either for better performance, or because certain resources are available in one stage but not another. Unfortunately, the various web programming languages make it difficult to spread computation over more than one stage. This is a tremendous opportunity for multi-stage languages in general, and for MetaOCaml in particular.

We present the design of MetaOCaml Server Pages. Unlike other languages in its genre, the embedded MetaOCaml code blocks may be annotated with staging information, so that the programmer may safely and precisely control which computation occurs in which stage. A prototype web server, written in OCaml, supports web sites with both static and dynamic content. We provide several sample programs and demonstrate the performance gains won using multi-stage programming.

1 Motivation

Modern dynamic web sites support many features for user collaboration and personalization. To provide such services, web sites contain custom computer programs, often written in one of a family of programming languages that have grown up around (or been adapted for) the web.

There is at least one dictum of program design that we cannot escape on the web: *performance matters.* As a web publisher, visitors are your livelihood. But will your servers and scripts be ready for the day that your site is featured on prime time television, or on `slashdot.org`? If tens of thousands of potential users drop by to find a sluggish (or dead) server, most of them will never return [11].

This happens so often to sites featured on Slashdot—a "news for nerds" discussion site—that it has come to be known as the *Slashdot effect:* "a site that might be designed to handle a few hundred hits per day can suddenly find itself handling that many a second." [22] Although some ISPs have bandwidth limitations, Slashdot creator Rob Malda says most sites that fail suffer from poor planning and architecture:

> Anybody who has a pretty good understanding of web design [has] done a good job of learning what information to cache [and] what needs to be pre-generated. So when you're actually loading a page, even if it's a complicated page that looks dynamic and custom, on the back end of that, what they're really doing is putting together a bunch of puzzle pieces that have been pre-generated, and making the simplest, quickest decisions they possibly can. [22]

Malda's observation points (informally) to the idea of *staging* the computation performed by a web service. Indeed, web publishing is an application area that is naturally staged: [1]

(1) Content (text, images, programs, etc.) created off-line is uploaded to the server—the *publish* stage;
(2) A user's browser requests content, which is transferred from server to client—the *serve* stage; and finally
(3) The content is rendered within the user's browser—the *display* stage. [2]

At *each* stage there is an opportunity for computation to take place. [3] Consider the example of a conference calendar, such as the one illustrated in figure 1. After specifying your areas of interest (perhaps using the ACM classifications), the server delivers a table of matching conferences, with dates, locations, deadlines, and links to conference web sites. Events remain in the table until a few weeks after they occur, but deadlines that have passed are marked in red. You may click on any column header to change the sort order. The next time you visit, the server remembers your preferences. Perhaps it even sends you email to remind you of upcoming submission and registration deadlines.

Now, how might this conference service be staged? Can anything be computed off-line (at the *publish* stage)? Yes: since this page is probably part of a much

[1] The literature on meta-programming has yet to acknowledge web services [2] as a potential application area, although Sheard [18] mentions *mobile code.* Analysis of other related work is in section 6.
[2] Increasingly in modern web applications, the rendered content is *interactive*—responsive to user input without a round-trip to the server.
[3] Nørmark [15] recognized these three stages as binding times, calling them *generated, calculated,* and *dynamic* documents, respectively.

event	deadline	location	date	description
ICALP	26 Jan 2004	Finland	12 Jul 2004	Automata, Languages, and Programming
ICLP	22 Feb 2004	France	6 Sep 2004	Logic Programming
OOPSLA	19 Mar 2004	Vancouver	24 Oct 2004	OO Programming, Systems, Languages, and Apps
GPCE	19 Mar 2004	Vancouver	24 Oct 2004	Generative Prog. and Component Engineering
ICFP	1 Apr 2004	Utah	19 Sep 2004	Functional Programming
PEPM	4 Apr 2004	Verona	24 Aug 2004	Partial Evaluation and Program Manipulation
PPDP	4 Apr 2004	Verona	24 Aug 2004	Principles and Practice of Declarative Prog.

Fig. 1. A simple conference calendar web service.

larger site (whose structure does not change every day), the menus and other navigation aids can be laid out in advance. We will not know which conferences to display until the user presents some identification (in the form of a 'cookie'), but since the conference data change infrequently, it may help to prepare the text of each row in advance.

During the *serve* stage, we look up the user's topic preferences, and ship out just the matching rows. If we delay sorting the table until the *display* stage, then the user ought to be able to adjust the sort criteria without any further communication with the server. What about marking past dates in red? If this is also delayed until display, then the code could be cached client-side for long periods of time, yet still behave dynamically.

At this point, we should emphasize the importance of *profiling* in developing scalable web services. This particular design for the conference calendar may not be optimal, depending on the number of entries and the relative speeds of the CPU, memory, database, network, and disk. Rather, our aim is to provide a single language in which the various staging possibilities can be expressed naturally.

This approach is in stark contrast to the status quo, where each system targets one stage only. The Website Meta Language [4] is an "off-line HTML generation toolkit" designed for the *publish* stage. But many other programs (and countless ad-hoc scripts) spit out HTML pages: LaTeX2HTML, for example. Google reports surprisingly many programs [5] for creating family tree web sites from genealogy database files; these also count as publish-stage tools.

The *serve* stage is well-served by the "server page" languages, including JSP,

[4] `http://www.thewml.org/`
[5] `http://google.com/search?q=gedcom+generate+html`

ASP, and PHP. The Common Gateway Interface (CGI)[6] addresses the *serve* stage, as do the embedded interpreters (such as `mod_perl` and `mod_python` for Apache) that exist to ameliorate some of the overhead of CGI.

There is, relatively, a paucity of languages that operate client-side (*display* stage), probably due to the difficulty of securing an installed base of interpreters. JavaScript, Java, and Flash applets are notable exceptions.

Imagine implementing the conference calendar, as conceived above, using currently deployed technology: a Perl script outputs a PHP page which embeds JavaScript! Values are passed from one stage to the next as strings, and the programmer must manage all the quoting and persistence issues by hand.

Strictly speaking, these languages are not exclusively confined to the stages that I have indicated: Javascript can be run server-side, PHP can be run offline, and so on. Nevertheless, migrating code between stages is hard, and the need for quoting and persistence are practically show-stoppers. For comparison purposes (further described in section 5), we staged some code using PHP. To achieve persistence of composite variables from one stage to the next, it contains gems like this:

```
<?= "<? \$list = unserialize(\"".
    addcslashes(serialize($list),'"').
    "\"); ?>\n" ?>
```

where the `serialize` library function occurs in stage one and the `unserialize` in stage two. Notice that the `$` preceding the first occurrence of `list` is quoted, but the second occurrence is not. The `addcslashes` function is needed in case the serialized representation contains special characters (such as double quotes or newlines) that would be misinterpreted by the PHP parser in the next stage. Interpreting the stage-one program guarantees nothing about the well-formedness of the stage-two program (generated as a string).

We present "MetaOCaml server pages," a new domain-specific language for web applications programming. It leverages the staging annotations and static typing of MetaOCaml [1, 19] to provide safe and precise control over the first two stages. (We leave further consideration of the display stage as future work.) The system is implemented as two components: a translator transforms the server page language into a MetaOCaml module, which then can be incorporated into our multi-threaded HTTP/1.1 server (also written in MetaOCaml). The scalability gained by staging certain applications is stunning: in section 5 we describe a directory browsing service where staging yields a factor of 30 improvement in throughput. The unstaged version would certainly succumb to the Slashdot effect.

[6] `http://www.w3.org/CGI/`

The next section sketches the design and translation of MetaOCaml server pages, and section 3 includes some non-trivial examples. The server implementation is decribed in section 4. Performance and scalability are discussed in section 5.

2 Design

The general idea of a *server page* language is that we write HTML by default, and embed code <? like this ?>. PHP programmers are familiar with this syntax for embedding code, but in our case, the code itself is written in MetaOCaml. Here is a trivial MetaOCaml server page:

```
<i>Hello,</i> <? failwith "Nice try!" ?> world.
```

and its output:

Hello, **Unhandled exception: Failure("Nice try!")**

The OCaml function failwith raises a Failure exception containing the provided message. If a code block raises an exception, the message is sent to the client's browser in bold-face, and the rest of the page is aborted.

In this example and throughout this paper, a sans-serif font is used for embedded MetaOCaml code, with **bold sans** reserved for keywords and code delimiters. A `typewriter` font is used for MetaOCaml character strings. The regular serif font is used for plain text and HTML within the server page, and for comments within the MetaOCaml code blocks.

A very common use of code blocks is to print out (i.e., send to the browser) the result of evaluating some expression. The syntax <?= e ?> is designated for this task; e must have type string. Alternatively, messages may be formatted with sprintf by placing the format string immediately after the code delimiter. [7]

```
<?= String.make 8 '.' ?> &pi; &divide;
<? "%03d is %.4f" 8, 3.14159 /. 8.0 ?>
```

The output is:

..... $\pi \div 008$ is 0.3927

[7] In a departure from the standard OCaml syntax for sprintf, the arguments are comma-delimited. This way, fewer parentheses are required when using the escape or lift operators on the arguments.

One more kind of code delimiter is used for *declarations;* these are lifted to the top of your program, and evaluated during the publishing stage:

```
<?^ open Unix let cwd = stat "." ?>
Permissions on current directory are <? "%04o" cwd.st_perm ?>
```

Output:

```
Permissions on current directory are 0755
```

Once published, this output will never change! The stat call is executed only once (because it is in a declaration block), not on each request. This is already a rudimentary kind of staging, but with the annotations of MetaOCaml, we will gain both flexibility and safety, as we'll see in the rest of this section.

2.1 Review of staging annotations

MetaOCaml augments OCaml with just three annotations, to indicate how programs are to be staged. Brackets .<e>. construct future-stage computation. The code within is not executed in the current stage of computation, but just returned as a code value that can be run later.

Within brackets, the splice or escape operator .~e may appear. It interrupts the code construction to evaluate the expression e (in the current stage) and splice its result into the future-stage computation. Thus, e is required to evaluate to code of the proper type.

To compute and splice in a regular (non-code) value, we define a function **let** lift x = .<x>.; this takes any value and turns it into code. We use it like this: .<2 * .~(lift(3+4))>. The addition is performed immediately (because it is escaped), and the result is spliced into the code, producing .<2 * 7>.

Finally, there is an operator .! to execute constructed code. Applying it to the example above, .! .<2 * 7 >. produces 14.

2.2 Translation to MetaOCaml

To see how all this works, consider how a MetaOCaml server page is translated into a proper MetaOCaml program, to be executed at publish time. Since the program is executed before any browser has requested the page, it cannot directly return or output HTML. Instead, it will construct and return a code object which is subsequently run on each request (serve stage). The third (display) stage proposed in section 1 is not yet supported by this design.

```
<? pragma args a b c ?>
<?^ declarations ?>
<? statements ?>
<?= string_to_be_printed ?>
Regular text.
<? "format string" d, e, f ?>
<?^ more_declarations ?>
<? let x = expression ?>
<? more_statements ?>
Bye!
```

Fig. 2. `trans.meta`. This file demonstrates the various kinds of code blocks.

```
module Trans = struct
let lift x = .<x>.
 declarations
 more_declarations
let page a b c = .<fun req puts →
let arg = Request.arg req in
 statements ;
puts ( string_to_be_printed );
puts "Regular text.\n";
Printf.kprintf puts "format string" ( d) ( e) ( f ) ;
 let x = expression in
 more_statements ;
puts "Bye!\n";

>.
end
```

Fig. 3. `trans.ml`. An automatic translation of the page in figure 2.

Figure 2 shows a sample MetaOCaml server page, and figure 3 contains its translation.

Declaration blocks have been lifted to the top; any side effects contained there are executed when the page is published. The constructed code begins on line 5.

The a b c represent publish-stage arguments (the names are specified with pragma args), whereas req and puts are (fixed) serve-stage arguments. req encapsulates the HTTP request details, including the headers and query arguments. puts is a function, provided by the server, to transmit text across the network to the user's browser. The library function Request.arg of the type request→string→string option looks up the string value of the request parameter with a given name. The option type constructor permits the function to

return None if there is no matching parameter in the HTTP request. On line 6, arg is defined as a short-cut for retrieving a parameter by name. Since req and puts are serve-stage arguments, it is incorrect to use them in the *publish* (first) stage, and indeed the MetaOCaml type system prevents this. These, along with the lift function defined near the top, are essentially primitives from the point of view of the server page code. [8]

2.3 Staged code blocks

Now that we understand how the server page is assembled into a staged program, the effects of adding MetaOCaml staging operators to our pages should be predictable. Below is another example using stat, this time to display the size of some text file on the server. With the serve-stage argument `unit`, we can specify whether the size should be expressed in bytes (the default), kilobytes, etc.

```
<?^ open Unix ?> <?! let st = stat "robots.txt" in
    let sz = float_of_int st.st_size in
    let (amt, unit) = match arg "unit" with
          | Some "M" → (sz /. 1048576., "M")
          | Some "k" → (sz /. 1024., "k")
          | _        → (sz, "") ?>
<?! "%.1f%s" amt, unit ?>
```

If this text file does not change frequently (and reporting outdated information is no problem), the stat and float_of_int calls could be lifted into the declaration block, as with the permissions example. Furthermore, we may also use lift and the splice operator to perform the divisions in advance, even though they are underneath the **match** (which cannot happen until the serve stage):

```
<?^ open Unix let st = stat "robots.txt"
    let sz = float_of_int st.st_size ?>
<?! let (amt, unit) = match arg "unit" with
      | Some "M" → (.~(lift(sz /. 1048576.)), "M")
      | Some "k" → (.~(lift(sz /. 1024.)), "k")
      | _        → (sz, "") ?>
<?! "%.1f%s" amt, unit ?>
```

[8] Here are a few finer points about the translator: it discards newlines that immediately follow code blocks (otherwise, figure 3 would be dotted with puts `"\n"` statements). It automatically appends a semi-colon or the **in** keyword to code blocks, as required (see, for example, lines 7, 11, and 12 in figure 3.) Finally, because the translator partially parses the OCaml code blocks, it is not confused by code delimiters within strings and comments, nor by other uses of < and > as operators.

Still, the printf conversion is performed at serve time, so maybe it is best to pre-generate that as well:

```
<?^ open Unix open Printf
    let st = stat "robots.txt"
    let sz = float_of_int st.st_size
    let fmt d u = lift(sprintf "%.1f%s" (sz/.d) u) ?>
<? match arg "unit" with
      | Some "M" → puts .~(fmt 1048576. "M")
      | Some "k" → puts .~(fmt 1024. "k")
      | _        → puts .~(fmt 1. "") ?>
```

Now, all that remains to execute at serve time is to check the dynamic unit argument and spit out one of three pre-generated strings. The generated code block looks something like this (substantially cleaned up from the MetaOCaml pretty-printer, with references to persistent values resolved in-place):

```
.< fun req puts → let arg = Request.arg req in
    (match (arg "unit") with
     | Some ("M") → (puts "0.0M")
     | Some ("k") → (puts "2.4k")
     | _ → (puts "2458.0")) >.
```

To handle more complex situations, code blocks may be constructed conditionally and recursively. Here is an example that prints a count-down, but removes the loop overhead by expanding to a sequence of 99 puts statements.

```
<?^ open Printf
   let rec count puts i =
     if i = 0 then .<() >.
     else .<(.~puts .~(lift(sprintf "%d<br>" i));
             .~(count puts (i−1))) >.
 ?>
<?.~(count .<puts>. 99) ?>
```

The brackets around the puts on the last line are necessary, because count splices the puts call into the code it generates. Omitting the brackets would result in a compile-time type error.

These tiny examples suggest the ease with which the 'boundary' between the stages can be adjusted, just by tweaking the staging annotations. Moreover, the static typing of MetaOCaml ensures in advance that our programs generate type-correct code only. The examples also illustrate some of the common uses of the escape operator, for which we developed syntactic sugar. The first set of code blocks we define use the tilde character to indicate that some part of the enclosed code is escaped:

```
<?~ a ?>            ⇝  <? .~( a ) ?>
<?~= b ?>           ⇝  <?= .~( b ) ?>
<?~ let x = c ?>    ⇝  <? let x = .~( c ) ?>
<?~ "fmt" d, e, f?> ⇝  <? "fmt" .~(d), .~(e), .~(f) ?>
```

where the a has type unit code; b has type string code; and the types of d,e,f match the format string.

It is also common to use the escape with lift; this means we are computing a value immediately and splicing it into the code. For these blocks, we use the ! character:

```
<?! a ?>            ⇝  <? .~(lift( a )) ?>
<?!= b ?>           ⇝  <?= .~(lift( b )) ?>
<?! let x = c ?>    ⇝  <? let x = .~(lift( c )) ?>
<?! "fmt" d, e?>    ⇝  <? "fmt" .~(lift(d)), .~(lift(e)) ?>
```

In these cases, the expressions should not have code types: a simply has type unit, b has type string, etc. This is the only difference between the ~ and ! variants; both are evaluated in the first stage. We will see examples of most of these blocks in section 3 and appendix A.

2.4 Publish-stage arguments

MetaOCaml server pages support publish-stage arguments, as demonstrated by the identifiers a b c in figures 2 and 3. The programmer specifies the pattern to be used with a <? pragma args ... ?> declaration, which may appear anywhere in the code. Any identifiers following args in the pragma declaration will become publish-stage parameters.

With publish-stage parameters, the page may be instantiated in countless ways. Continuing the count-down example, we could map the URI `/longcount` to a code block generated with an argument of 99, while `/shortcount` refers to code with the argument 9. Both are generated from the same server page.

```
<? pragma args n ?>
<?^ open Printf
  let rec count puts i = if i = 0 then .<()>.
      else .<(.~puts .~(lift(sprintf "%d<br>" i));
              .~(count puts (i-1)))>. ?>
<?~ count .<puts>. n ?>
```

```
val preamble: string → string
  (* Generate a standard HTML header, including a title block
     composed from the given string. *)
val navbar: string → string
  (* Generate the navigation bar, given a URI denoting
     the current page. *)
val postamble: string
  (* The page footer. *)
```

Fig. 4. These functions help define the standard site layout.

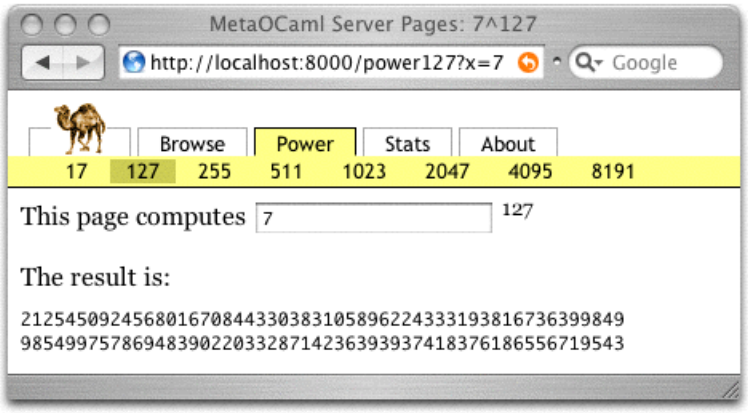

Fig. 5. Web browser displaying the result of 7^{127}.

Mapping from URIs to instantiated MetaOCaml code is, for now, left as an implementation detail (see section 4).

3 Examples

To explore the expressiveness of this design, we now look at a series of web services, organized to comprise a small web site. To give the services a similar look, we developed a site-wide style sheet, and an OCaml function to generate a navigation bar from a hierarchical list of page titles and links. The page layout functions in figure 4 may be invoked at any stage. Naturally, if the page title includes a dynamic argument, then preamble will have to be delayed until the serve stage.

```
<?^ open Num (* for arbitrary-precision arithmetic *)
  let width = 54
  let rec wrap puts s = (* wrap s into a fixed-width block *)
    if String.length s ≤ width then puts s else
      (puts (Str.string_before s width); puts "\n";
       wrap puts (Str.string_after s width))
  let is_zero = eq_num (Int 0)
  let square x = .< let z = .~x in z */ z >.
  let rec power n x = (* staged power function *)
    if is_zero n then .< Int 1 >. else
    if is_zero (mod_num n (Int 2)) then square(power (n//Int 2) x)
    else .< .~x */ .~(power (n -/ Int 1) x) >. ?>
<? pragma args y ?>
<?! let y' = string_of_num y ?>
<? let x' = match (arg "x") with Some v → v | None → "2" ?>
<?= preamble(x'^"^"^y') (* Output begins here *) ?>
<?!= navbar("/power"^string_of_num y) ?>
<form method='get'> This page computes
  <input name='x' type='text' value='<?= x' ?>' size='20'/>
  <sup><?= y' ?></sup> </form>
<?~ let result = power y .< num_of_string x' >. ?>
<p>The result is:
<pre><? wrap puts (string_of_num result) ?></pre></p>
<?= postamble ?>
```

Fig. 6. `power.meta`. The staged power function as illustrated in figure 5.

3.1 The ubiquitous power function

Judging from its prevalence in the multi-stage programming literature, we are certain that millions of grateful users would subscribe to an online service capable of computing the *power* function. The screen shot in figure 5 illustrates how it works. The navigation bar shows the exponents for which code has been pre-generated. [9] After selecting the exponent, the user types the base into the form and presses return. The result is computed using the arbitrary-precision Num module of the OCaml library. The complete script appears in figure 6.

The user's input is converted to a number relatively late in the script (line 21). If num_of_string generates an exception (perhaps because the user typed non-numeric text into the box), the navigation bar and form will have already been output before the error message appears.

[9] In the current implementation, it is not possible for the user to request other exponents once the server is running. See section 4 for an explanation.

In section 5, we will measure the impact of staging on the scalability of this application. To derive the un-staged version for comparison, we simply replace `<?~` and `<?!` blocks with plain `<?` and remove all other brackets and escapes from the code in figure 6.

3.2 Directory browsing

Now we consider a more substantial example. Many web servers can be configured to permit clients to browse directories. The web server generates, on the fly, an HTML page containing the names and attributes of (and hyperlinks to) all the files in whatever directory is specified by the URL. In Apache, the `mod_autoindex` module provides this feature. ViewCVS is a more complex example of the same idea; it allows remote users to browse a CVS repository with a standard web browser.

This kind of service can be fairly resource intensive; each HTTP request is likely to generate dozens of system calls and disk accesses. If the directory is viewed more often than it is changed, then it makes sense to cache or pre-generate the pages. Although it is simple enough to write a script to generate static directory pages off-line, what if we also want dynamic behavior, such as user-controlled sorting and filtering?

Our implementation not only lists files in a given directory, but displays their MD5 checksums, renders their sizes in human-readable form ('1.2M' rather than '1194822'), and colors their names based on their extension or file type. We also support dynamic (serve time) customization: the user may specify a regular expression for filtering, and select one of 5 criteria for sorting. The screen shot in figure 7 shows the result of browsing the source directory of the server itself.

This service is a bit like the conference calendar proposed in section 1. We pre-generate *everything* that does not rely on serve-time parameters. The stats and MD5 sums of the files are collected *once,* in the publish stage. The text of each possible row is prepared in advance. Once the user's request is made, we filter and sort the list, then output the pre-generated text of each remaining row. The files need not be opened or even `stat(2)`ed while the user is waiting. Appendix A contains the complete script, with documentation.

Much of the code is unaffected by staging; helper functions (that sort according to the selected criterion, format the human-readable sizes, and collect the file information) are oblivious to the stage in which they are run. Therefore, the staged code comprises a relatively small portion of the entire program: mainly the function list_files and its call site.

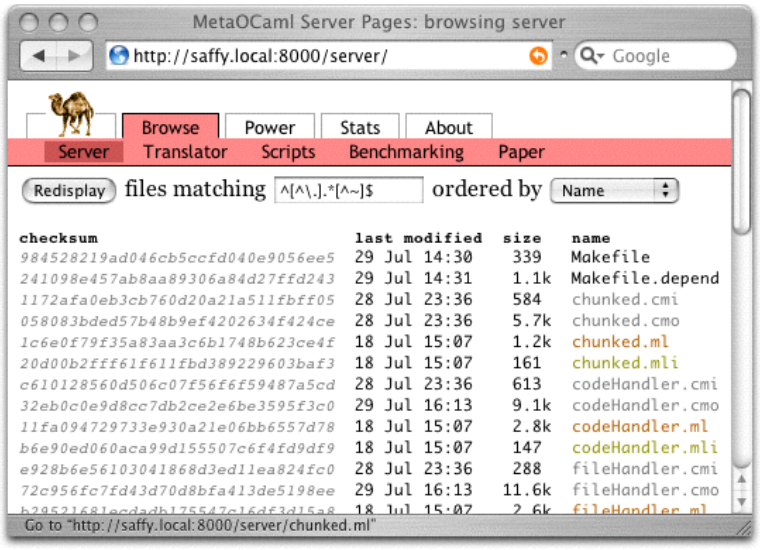

Fig. 7. Browsing the server source code directory.

To understand the staging technique, it may help to examine the output of just the first stage of computation. The code block in figure 8 is cleaned up from the MetaOCaml pretty-printer, with references to compiled values substituted in-place. The list is formed by testing filenames (in reverse alphabetical order) against the compiled regular expression rc, prepending only those that match. The file info record for each file has already been prepared and appears in the code as a value.

Because the file information is compiled into the code, any changes to the file system following the first stage of execution will *not* automatically appear on the web interface. To see the latest files, we need to re-run stage one. This cannot be done from within the MetaOCaml server page itself, but we have programmed the server to regenerate pages whenever the '!' character is appended to the URI. The directory browser pages feature a 'Regenerate' button at the bottom which will bring them up to date by running the publish stage code again, and caching the result for subsequent requests.

3.3 Server introspection

Some web servers can be configured to display their status in response to certain URIs (such as `/server-status` on Apache). We programmed a few status services in MetaOCaml. The screen shot in figure 9 displays garbage collection statistics from the OCaml `GC` module. The only thing pre-generated here is the navigation bar.

```
.<fun req puts →
  let arg = Request.arg req in
  puts "<html>\n<head>\n<title>MetaOCaml. . .";
  let (re,rc) = match arg "re" with
        None → default_re | Some r → (r, Str.regexp r)
  let list =
      try ignore(Str.search_forward rc "timeStamp.mli" 0);
          [{name="timeStamp.mli", prn=". . .", ...}]
      with Not_found → [] in
  let list =
      try ignore(Str.search_forward rc "timeStamp.ml" 0);
          {name="timeStamp.ml", prn=". . .", ...} :: list
      with Not_found → list in
  let list =
      try ignore(Str.search_forward rc "timeStamp.cmo" 0);
          {name="timeStamp.cmo", prn=". . .", ...} :: list
      with Not_found → list in
    :  (* and so on, for the rest of the files. *)
  let ord = match arg "ord" with
        None → "name" | Some o → o in
  let list = sort_by ord list
  puts "<form method='get' . . .";
   :  (* etc. *)
>.
```

Fig. 8. Generated code for directory browser; see also appendix A.

4 Implementation

The implementation consists of two parts: a translator and a web server. The translator transforms the server page syntax into plain MetaOCaml, as illustrated in figure 3. It recognizes all the server page blocks defined in section 2 but does not completely parse the MetaOCaml code contained within them. Instead, it recognizes just enough of the keywords and delimiters to decide whether to add semi-colons or '**in**' after each block. This approach makes the translator fairly robust to minor changes in OCaml syntax.

The disadvantage of this method is that very few syntax errors (and no type errors) are detected by the translator itself. So, errors reported by MetaOCaml are displayed in terms of the translated program, not the source program.

The server component is much more complex. We implemented the essential parts of the HTTP/1.1 specification as a multi-threaded OCaml program. Upon receiving a request for some URI, the server uses a *chain of responsibility* [7] to determine how to handle it. The chain is specified as a parameter to

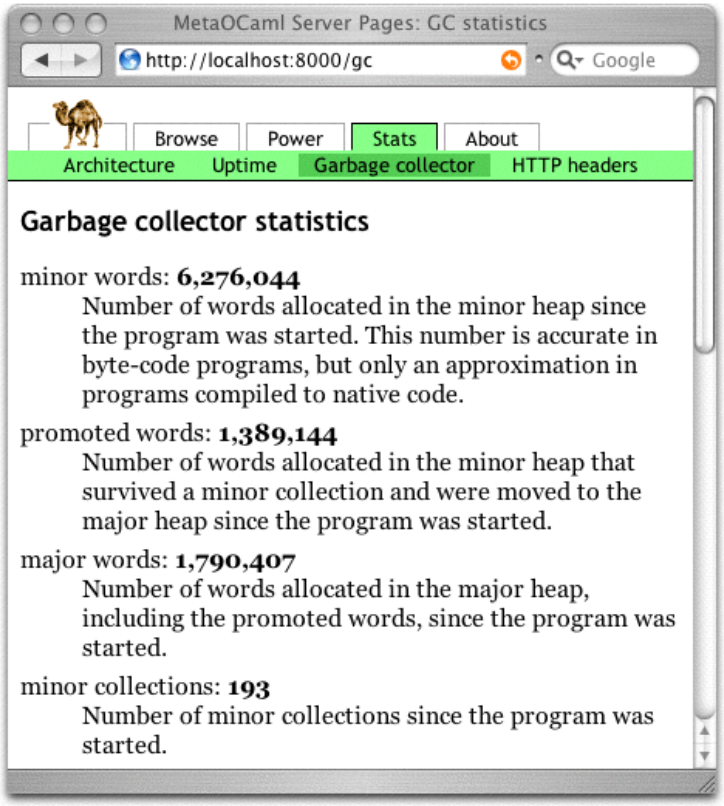

Fig. 9. Garbage collector statistics, from the OCaml gc module.

the server. Two primary handlers are provided: FileHandler and CodeHandler.

The FileHandler takes a file system path as a parameter, and then interprets each URI as a file name relative to that path. If such a file exists, it sends it verbatim to the client. If not, it passes the responsibility on to the next handler in the chain.

The CodeHandler is provided with a dictionary to map URIs to code values. It looks up the URI in the dictionary, and if a match is found it invokes that code. For now, the map is hard-coded at build time and the code values are already in memory when the server starts accepting requests. This is because MetaOCaml does not currently permit programs to read and write code values to a file.

Here are the types of the page code and the map, along with the signature for the CodeHandler:

```
type page = Request.req → (string → unit) → unit
type map = ((unit → page) * page ref) StringMap.t
val run : map → Server.handler
```

page is the type of the code that is constructed by each server page; recall that the page takes two serve-time parameters: one encapsulating the HTTP request, the other is the puts function for sending text back to the client.

Data structures of type map tell the server which pages are mapped to which URIs. The range of the map is a pair: the first component (of type unit→page) is a function that will re-run stage one computation (using the .! operator internally); the second component (of type page ref) is a cell where the latest page is cached.

The unit→page formulation is a work-around for what might otherwise be expressed as page code, and run from within the CodeHandler. Unfortunately, code values in MetaOCaml must remain polymorphic to run them; the type is actually ('a, page) code, for all 'a. This is traditionally problematic in ML: we can define a polymorphic data structure, but not a data structure containing polymorphic values. The work-around we used captures the polymorphic code value in a closure, which is then added to the map. Another possibility is to use the limited form of explicit polymorphism supported in OCaml, with the syntax {f : 'a. ('a, page) code}. [10] To use code values more directly, we would need rank-2 polymorphism; work by Garrigue and Rémy [8] is headed in that direction.

5 Performance

In this section, we describe the performance characteristics of the prototype, focusing in particular on the benefits of staging various web services. The single most important metric for evaluating web server performance is *throughput:* the number of requests successfully answered per unit of time. To establish a baseline, we first tested the throughput of our custom OCaml HTTP server delivering chunks of static data of various sizes, up to 64k bytes. Measurements were taken on an otherwise idle 1.8GHz Intel Xeon workstation [11] running Linux 2.6. We used ab, the Apache HTTP server benchmarking tool, to issue requests from 8 threads simultaneously for 30 seconds. [12]

The baseline results are shown in figure 10. In the "OCaml FileHandler" series, the files were treated just as static files, opened on disk, and copied out to the socket. There are two sets of results for the FileHandler: one compiled as native code, and the other compiled as byte code. A native code compiler for MetaOCaml is not yet available. Therefore, most comparisons in this section

[10] Thanks to an anonymous reviewer for pointing this out.
[11] with 512kB cache, 768MB RAM, and Ultra160 SCSI
[12] invoked like this: `ab -k -t 30 -c 8` *url*

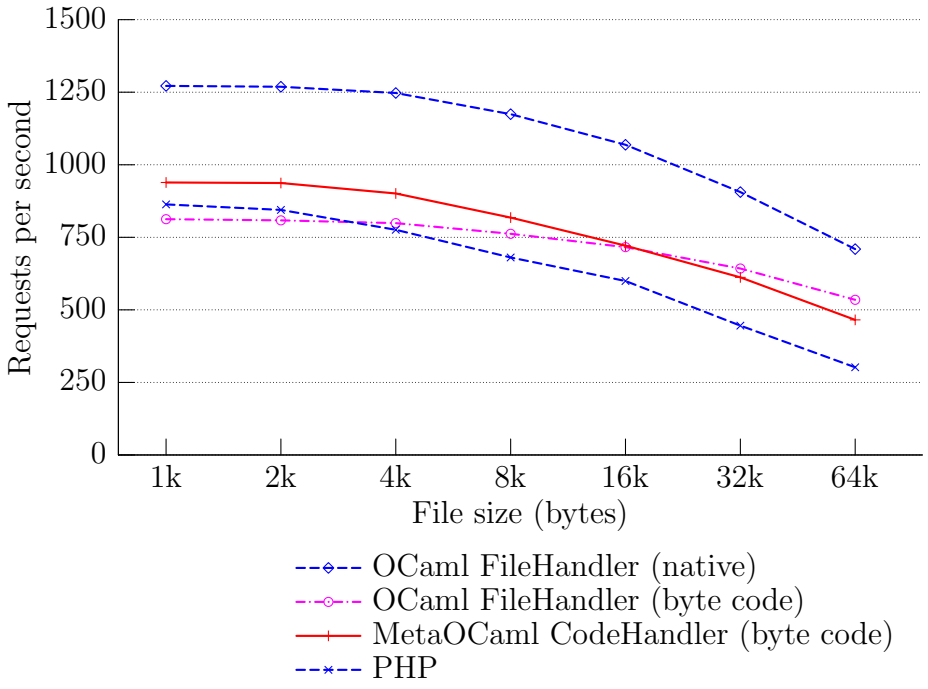

Fig. 10. Throughput for static pages. Note the logarithmic scale on the x axis.

will rely on byte code. The native code results we are able to obtain at this time suggest how much improvement we can expect once native code is an option.

For the "MetaOCaml CodeHandler" series, the same set of files were treated as MetaOCaml Server Pages, and thus translated (in advance) into one big `puts` statement, to be executed (as byte code) by the `CodeHandler` module. The only reason the code beats the (byte code) FileHandler in the beginning is that all code pages are already loaded into the server's memory on startup (to work around a limitation of the prototype—see section 4), but the FileHandler must read files from disk each time.

Figure 10 includes comparable results for PHP, the popular server-side computation system.[13] Here, we gave the same static data files the extension `.php`, so that Apache would treat them as PHP code, even though they have no `<?` code blocks `?>`. We omitted the results for Apache serving static files, because they are way off scale: Apache handled an astounding 4,438 hits per second for the 1k file, and 1,813 for the 64k file. Apache is heavily optimized for serving static files: apart from caching, it uses the special `sendfile(2)` system call for zero-copy file transfer from kernel space [21]. Although our prototype is no match for Apache on static files, the overhead for interpreting code seems no worse than that of PHP.

[13] libphp4.so (version 4.3.10) loaded into Apache 1.3

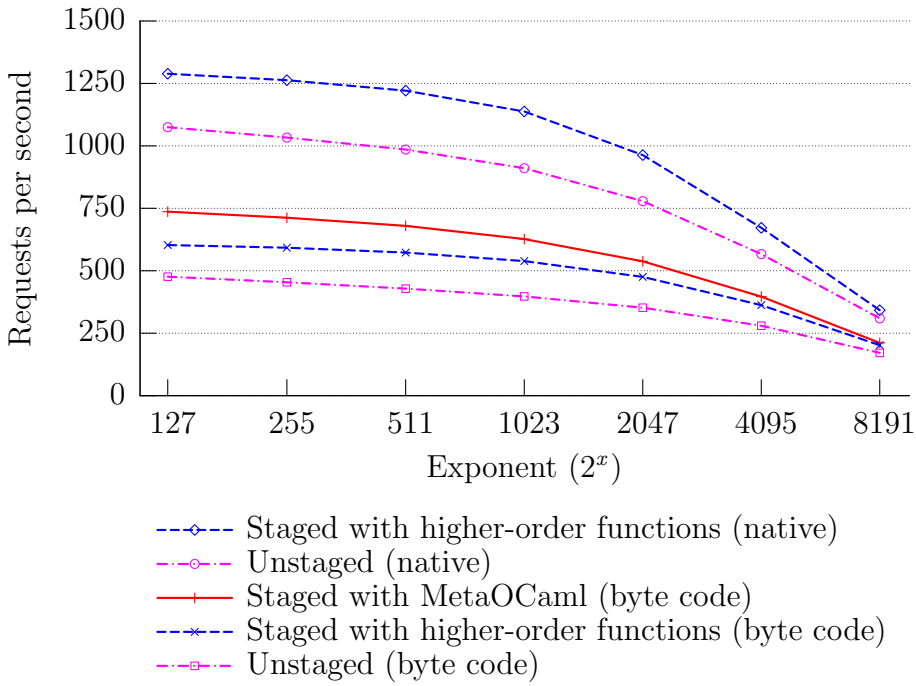

Fig. 11. Staged versus unstaged power function. The server computes 2^x.

```
let rec power n =
  if is_zero n then fun x → Int 1
  else if is_zero (mod_num n (Int 2))
       then let f = power (n // (Int 2)) in fun x → square (f x)
       else let g = power (n −/ (Int 1)) in fun x → x */ (g x)
```

Fig. 12. Staging the power function using higher-order functions.

Using the same methodology, we now consider the performance of the staged and unstaged power functions; see figure 11. Since we are comparing staged vs. unstaged (and not native vs. byte code), these results need to be interpreted in two distinct groups. The top two lines are native code, while the bottom three lines are byte code. Start with the byte code.

The throughput for the pages staged with MetaOCaml is about 30% higher than the unstaged version in the beginning, but as the exponents increase (again, note the log scale in the graph) the gap narrows. In this program, staging removes the loop overhead, but the cost of the multiplications and conversion to a decimal string (which involves non-trivial divisions) are needed either way. Eventually, the cost of those operations dominates everything else. [14]

At the suggestion of an anonymous reviewer, we also tried staging using higher-order functions in OCaml instead of the code splicing features of MetaOCaml. The relevant fragment of code is shown in figure 12. It recursively builds up a

[14] The final result, 2^{8191} has 2,466 digits in base 10.

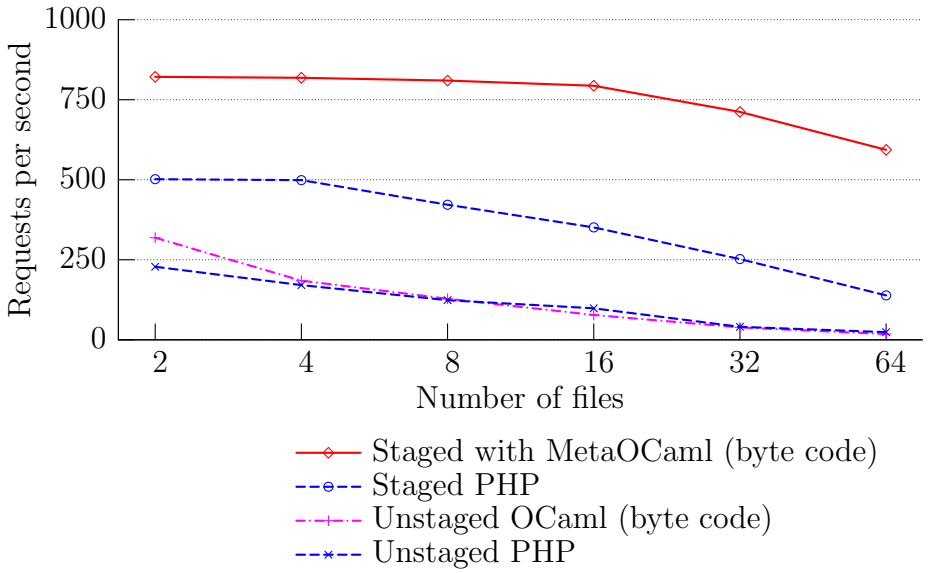

Fig. 13. Staged versus unstaged directory browsing, in MetaOCaml and PHP.

closure that represents the computation required to compute any number to the power `n`.

As one might expect, staging using higher-order functions is slightly worse than producing specialized code as in MetaOCaml, but better, of course, than no staging at all. Moreover, this version compiles easily to native code with the standard OCaml tools, and this improves its performance significantly. We do have reason to hope, however, that the native MetaOCaml version, once working, will ultimately perform the best.

Finally, we look at the performance of staged and unstaged directory browsing; see figures 13 and 14. Here, we created directories containing fixed numbers of files with random data. The average file size was 32k. The x axis shows the number of files in each directory.

Besides the staging factor, two implementation languages are compared: we implemented the same functionality in PHP. The 'staged PHP' version must be run first from the command line; this outputs a PHP script which is then run by the server. With PHP we must, as noted in section 1, manage the quoting and persistence by hand. (This program is the source of the horrible `serialize`/`unserialize` code shown on page 4.)

Directory browsing was the most realistic of the examples, and the benefit of staging is crystal clear. In browsing a directory of 64 files, the unstaged programs barely answered 18 requests per second. They would certainly succumb to the Slashdot effect—and compilation to native code made essentially no difference.

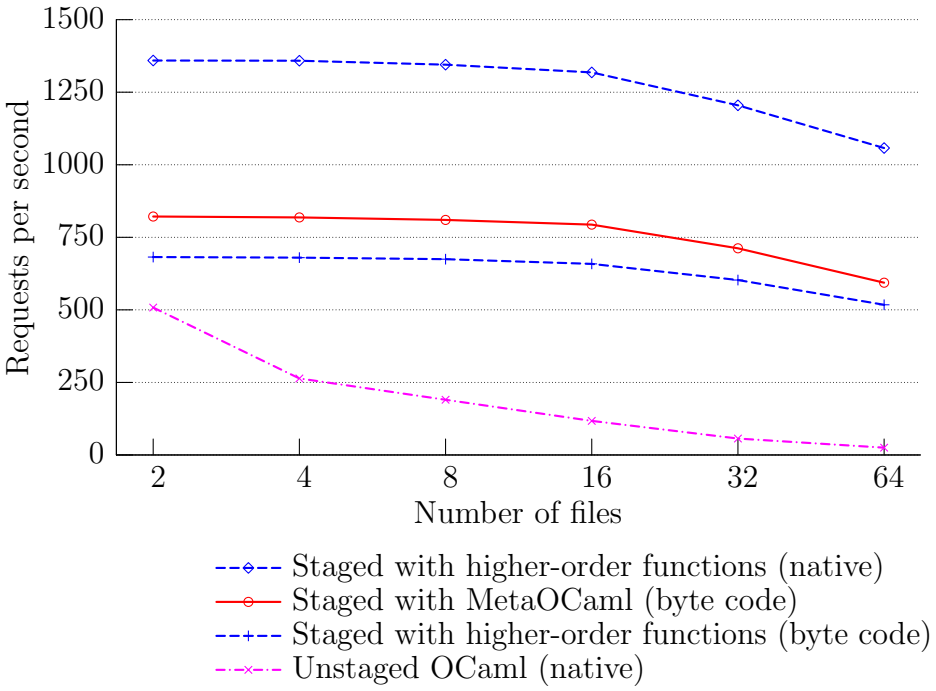

Fig. 14. Directory browsing staged with higher-order functions vs. MetaOCaml.

This is exactly the kind of page where the real work needs to be done in advance. But that does not mean it needs to be a completely static page, either. By carefully staging the computation, we gathered the file information in advance, yet still filtered and sorted the results on demand. The staged MetaOCaml directory browser answered more than 550 requests (for the same 64-file directory) per second.

The degradation in performance of the staged PHP version is most likely due to the fact that the stage two script must be *parsed* on each request; the size of the script is proportional to the number of files in the directory. With MetaOCaml, the code size is also proportional to the number of files, but the values are byte-compiled (and already in memory). At any rate, our goal is not to beat PHP on performance, but rather to gain the performance benefit of staging *without* the awkwardness of staging in a language (like PHP) that does not support it.

We also tried staging the directory service using higher-order functions, the same technique as described for the power function. Again, this did not quite match the performance of the MetaOCaml version (when comparing byte-code to byte-code). However, in figure 14, we can also see the substantial difference that compilation to native code can make. We expect that the native MetaOCaml compiler will enable even better throughput.

6 Related and future work

Most of the *server page* systems embed programs within web pages using similar techniques; examples include JSP, ASP, PHP, MSP [5] (based on SML), and AS/XCaml (based on OCaml). [15] Ours appears to be the first such language with explicit support for *staging* the computation. We inherited the staging annotations of MetaOCaml [1] and designed several new kinds of code blocks based on them.

There is much related work on using various features of modern programming languages to implement sophisticated web services. Nørmark [15] proposed writing web pages in Scheme using the Lisp Abstracted Markup Language (LAML), which essentially represents HTML documents as S-expressions. He distinguished three different binding times, when the Scheme program could be evaluated: off-line, page access time, or browse time (client-side). These correspond precisely to the three stages we identified in section 1, but his programs did not *transcend* different stages. It is possible to implement staged programming in Scheme using quasiquote, unquote, and eval [3]; this would lead to a similar capability, save for the difference between static and dynamic checking of the generated code.

Takebe and Yuasa [20] developed a partial evaluation technique for PHP. Their tool, PHP-Mix, reads standard PHP source code and performs binding time analysis. Then, it generates a semi-static script as output. It understands a significant subset of the PHP function library, including the database connectivity. Unfortunately, some properties of PHP—such as the lack of variable declarations and the single global scope—cause the analysis to be overly conservative, so it does not always achieve the optimizations we desire.

Queinnec [16] and Hughes [13] observed that multi-page web services could be implemented more naturally using continuations and call/cc (essentially, by treating the server and user as coroutines). This way, we can treat the entire service as one program—that suspends itself while waiting for user input—instead of developing each page as a separate program. This is a valuable technique, and appears to be orthogonal to staging; although it would be worth implementing both together to determine if there are unforeseen interactions.

Graunke et al. [9, 10] took up this idea and mixed it with other language features: first-class modules, preemptive threads, and *custodians,* for managing resource consumption. The result is a server that achieves lower overhead for dynamic services compared to Apache CGI (although the overhead of CGI is avoided by most modern dynamic services, by embedding interpreters for JSP, PHP, or similar in the server). Matthews et al. [14] compile such direct-style

[15] Application System Xcaml, by Alessandro Baretta. `http://www.asxcaml.org/`

interactive programs into CGI-style scripts.

Unlike more ad-hoc staging methods, MetaOCaml server pages guarantee the type safety for generated code up front. We do not, however, make any guarantees about the validity of the generated HTML. Elsman and Larsen [6], Wallace and Runciman [23], Hosoya and Pierce [12], and Ohl [16] leverage ML-like type systems to validate (X)HTML generators. Integrating their ideas into MetaOCaml server pages may permit validation of the generated document as well.

Our server page language is defined by translation into MetaOCaml. Unfortunately, this means that error messages refer to the translated code, not the original, embedded code. Camlp4 is a flexible pre-processor capable of modifying the concrete syntax of OCaml while maintaining usable error messages. [17] Formulating the server page syntax with Camlp4 would likely be an improvement over the current, ad-hoc translator.

In motivating MetaOCaml server pages, we observed that the computation associated with a web page naturally decomposed into three stages: publish, serve, and display. Our language, however, was designed to support just the first two stages. It is straightforward to imagine an extension to the third stage (since MetaOCaml itself has no constraints on the number of stages) but the implementation may be tricky. First, we must overcome the process boundary between server and client. We currently have no way to export MetaOCaml code blocks from the program that created them into another program. (On the server, we circumvented this limitation by running the publish and serve stages within one process.) Next, we need a browser capable of loading and executing MetaOCaml code. Rouaix [17] demonstrated a browser called MMM, written in Caml, that could run applets loaded as Caml byte-code. It may be possible to update his browser for use with MetaOCaml, but developing a plug-in that works with conventional browsers would be preferable.

One of the major shortcomings of our implementation is that all the pages must be loaded into memory when the server starts. Ideally, a separate tool would translate the server page source directly to byte-code, stored in an image on disk. Then the server would map URIs to these byte-code files, loading and executing them on demand. Whenever we want to re-run the publish stage, we could do so independently of the server.

Another interesting avenue is to incorporate work on *offshoring* [4]. An alternative *run* construct translates generated code blocks to lower-level languages for improved performance. In the domain of web services, it is conceivable that a MetaOCaml server page could then generate (via offshoring) a special-

[16] XHTML module. `http://theorie.physik.uni-wuerzburg.de/~ohl/xhtml/`
[17] Camlp4, by Daniel de Rauglaudre. `http://pauillac.inria.fr/caml/camlp4/`

ized server page in a more mainstream language, such as PHP, JSP, or—for client-side computation—JavaScript.

More substantial applications are needed to demonstrate both the performance gains and the expressiveness of this approach. A web service that uses a database and is spread over several pages would be more realistic. Queinnec [16] and Graunke et al. [9] use features of functional languages to express user interactions more naturally, given the stateless nature of HTTP. We believe that these techniques are orthogonal to the staging features provided by MetaOCaml.

Finally, although our HTTP implementation stands up to the Apache benchmarking tool, it is not likely to win any awards for reliability or flexibility. Our ideas would likely have greater impact if they were implemented as a module for a real server such as Apache or AOLserver. Beyond the concerns outlined earlier in this section, this is likely to be 'just' an engineering effort.

7 Conclusion

Web publishing is an important application domain that is naturally staged. Web programmers write staged programs, but they do it the old-fashioned way: one script outputs another as a string. This is a tremendous opportunity for multi-stage languages.

We presented the design of MetaOCaml server pages, a new domain-specific language for web applications programming. It leverages the staging annotations of MetaOCaml to provide safe and precise control over the each stage of the computation. We have shown the substantial benefits of this approach in terms of performance and expressiveness, although the prototype implementation suffers some limitations because it is unable to read and write generated code to a file.

Acknowledgments

Sincere thanks to the anonymous reviewers, for many helpful and enlightening recommendations on both the work and the presentation. Thanks to my M.S. student, Kiichi Takeuchi, for exploring some applications of these ideas, and for helping me understand the work of Takebe and Yuasa [20] (written in Japanese). Thanks to Walid Taha for organizing the MetaOCaml workshop, and to the rest of the MetaOCaml team for their development efforts.

References

[1] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. Conf. on Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 57–76. Springer, 2003.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. Technical report, World Wide Web Consortium, March 2001. URL `http://www.w3.org/TR/wsdl`.

[3] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[4] J. Eckhardt, R. Kaiabachev, and K. Swadi. Offshoring: Representing C and Fortran 90 in OCaml. In *Proc. First MetaOCaml Workshop*, 2004.

[5] M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Symp. on Practical Aspacts of Declarative Languages*, volume 2562 of *LNCS*, pages 74–91. Springer, January 2003.

[6] M. Elsman and K. F. Larsen. Typing XHTML web applications in ML. In *Symp. on Practical Aspacts of Declarative Languages*, volume 3057 of *LNCS*, pages 224–238. Springer, June 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155(1):134–169, 1999.

[9] P. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level languages. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 122–136. Springer, 2001.

[10] P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 238–252. Springer, April 2003.

[11] P. Greenspun. *Philip & Alex's Guide to Web Publishing*. Morgan Kaufmann, 1999.

[12] H. Hosoya and B. C. Pierce. XDuce: a typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[13] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

[14] J. Matthews, R. B. Findler, P. Graunke, S. Krishnamurthy, and M. Felleisen. Automatically restructuring software for the web. *J. Automated Software Engineering*, 11(4):337–364, October 2004.

[15] K. Nørmark. Programmatic WWW authoring usinch Scheme and LAML. In *Proc. Int'l. World-Wide Web Conf.*, 2002.

[16] C. Queinnec. The influence of browsers on evaluators; or, Continuations to program web servers. In *Proc. Int'l Conf. Functional Programming*, pages 23–33. ACM, 2000.

[17] F. Rouaix. A web navigator with applets in Caml. *Computer Networks and ISDN Systems*, 28(7–11):1365–1371, May 1996.

[18] T. Sheard. Accomplishments and research challenges in meta-programming. In *Proc. Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *LNCS*, pages 2–44. Springer, 2001.

[19] W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[20] Y. Takebe and T. Yuasa. A caching system for dynamic web pages using partial evaluation. *IPSJ Transactions on Programming*, 43(8), September 2002. IPSJ is the *Information Processing Society of Japan* (in Japanese).

[21] J. Tranter. Exploring the sendfile system call. *Linux Gazette*, 91, June 2003. URL `http://www.linuxgazette.com/issue91/tranter.html`.

[22] J. Turner. How to survive being slashdotted: Tips on how to survive a sudden increase in web traffic. *LinuxWorld Magazine*, 2(1), December 2003. URL `http://linux.sys-con.com/read/38293.htm`.

[23] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. Int'l Conf. Functional Programming*, pages 148–159. ACM, 1999.

A The directory browser

```
<? pragma args uri d ?>
```

The script takes two publish-stage arguments: uri is the server path of the generated page, and d is the filesystem path of the directory to browse.

```
<?^ open Printf open Unix ?>
```

We assemble the page title and navigation bar in the publish stage. These are then propagated as a string to be printed in the serve stage.

```
<?!= preamble("browsing "^d) ^ navbar uri ?>
```

The default regular expression is compiled in advance, but superseded if the re argument is provided in the HTTP request.

```
<?^ let default_re = let r = "^[^\\.].*[^~]$" in (r, Str.regexp r) ?>
<? let (re,rc) = match arg "re" with
        None → default_re | Some r → (r, Str.regexp r) ?>
<?~ let list = list_files d .< rc >. ?>
```

The above call to list_files occurs in the first stage, but it produces code to be executed in the second stage. The first stage stats all the files and computes MD5 sums. The next stage filters according to the regular expression, passed to list_files as code. The function list_files appears later in this script.

```
<? let ord = match arg "ord" with
        None → "name" (* default *) | Some o → o in
```

```
  let list = sort_by ord list in ?>
```

Sorting occurs entirely in the second stage. The function sort_by is also defined later in this script.

```
<form method='get' action=''>
<input type='submit' value='Redisplay' /> files matching
<input type='text' name='re' size='14' value='<?= re ?>' />
ordered by <select name='ord'>
<?~ ord_options .< puts >. .< ord >.
      ["name", "Name"; "ext", "Extension";
       "time", "Timestamp"; "size", "Size";
       "kind", "Kind"] ?> </select></form>
```

ord_options (defined later) generates the option tags for the drop-down menu.

```
<pre>
<?= header (* column heads *) ?>
<? List.iter (fun f→puts f.prn) list ?>
</pre>
<form method='post' action='<?= uri ?>!'>
<input type='submit' value='Regenerate' /></form>
<?= postamble (* page ends *) ?>
```

Recall: declaration blocks (like the one below) are lifted up before any of the page code; that is why we seem to use functions like list_files before defining them.

```
<?^ type fileinfo = { name: string; ext: string; kind: string;
      mtime: float; size: int; md5: string; prn: string } ?>
```

The list we intend to build includes not only file names, but all the file data and even the printed representation, all of it prepared in advance. The various sort orders use different fields of the above record type.

```
<?^ let cex f1 f2 = compare f1.ext f2.ext
    let ck f1 f2 = compare f1.kind f2.kind
    let cmt f1 f2 = compare f1.mtime f2.mtime
    let csz f1 f2 = compare f1.size f2.size
    let sort_by order list = match order with
     | "ext" → List.stable_sort cex list | "kind" → List.stable_sort ck list
     | "time" → List.stable_sort cmt list | "size" → List.stable_sort csz list
     | _ → list ?>
```

The following format string is used to generate the printable representation of each entry. The column headings are defined similarly.

```
<?^ let entry_fmt = format_of_string
"<span class='md5'>%−32s</span> %−13s %5s <a class='file' \
```

```
  href='%s'><span class='%s'>%s</span></a>%s\n"
let header = sprintf
 "<b>%-32s %-13s %-5s %s</b>\n"
 "checksum" "last modified" "size" "name" ?>
```

Generate 'human-readable' sizes:

```
<?^ let human_size n =
      if n < 1024 then sprintf "%4d " n
      else if n < 102400 then sprintf "%4.1fk" (float_of_int n/.1024.)
      else if n < 1024000 then sprintf "%4dk" (n/1024)
      else if n < 104857600 then sprintf "%4.1fM" (float_of_int n/.1048576.)
      else sprintf "%4dM" (n/1048576) ?>
```

Generate kind and symbol (like '–F' option of ls) from filename extension and/or file permissions.

```
<?^ let reg_kinds ext perm =
      let k = match ext with
        | "a"→"lib" | "cma"→"lib" | "cmi"→"obj" | "cmo"→"obj"
        | "ml"→"src" | "sml"→"src" | "mli"→"hdr" | "sig"→"hdr"
        | _→"" in
      let i = if perm land 0o111 = 0 then "" else "*" in
      match (k,i) with
        | ("", "*") → ("exe", "*") | other → other ?>
```

Gather all the info for file f in directory d.

```
<?^ let fileinfo d f =
      let path = Filename.concat d f in
      let st = stat path in
      let md5 = if st.st_kind = S_REG
        then Digest.to_hex(Digest.file path) else "" in
      let ext = try let i = String.rindex f '.' + 1 in Str.string_after f i
                with Not_found → "" in
      let (kind, indicator) = match st.st_kind with
        | S_DIR→("dir", "/") | S_FIFO→("fifo", "|")
        | S_BLK→("bdev", "") | S_CHR→("cdev", "")
        | S_LNK→("link", "@") | S_SOCK→("sock", "=")
        | S_REG→reg_kinds ext st.st_perm
      in let prn = sprintf entry_fmt md5 (TimeStamp.brief st.st_mtime)
                     (human_size st.st_size) f kind f indicator
      in {name=f; ext=ext; kind=kind; md5=md5; mtime=st.st_mtime;
          size=st.st_size; prn=prn} ?>
```

List filenames, in reverse alphabetical order:

```
<?^ let rc x y = - (compare x y)
```

```
    let rec read_all dh files = try read_all dh (readdir dh :: files)
        with End_of_file → closedir dh; List.sort rc files ?>
```

Walk through all the files and gather their information, then generate code to filter based on filename.

```
<?^ let list_files d re =
      let rec loop term files = match files with [] → term
        | name::files →
          .<let list = try ignore(Str.search_forward .~re name 0);
                           .~(lift(fileinfo d name)) :: .~term
                with Not_found → .~term
            in .~(loop .<list>. files)>.
        in loop .<[]>. (read_all (opendir d) []) ?>
```

Generate code to print the option tags, adding the selected attribute as appropriate.

```
<?^ let rec ord_options puts ord opts =
      match opts with [] → .<()>. | (tag,text)::opts →
      .<(kprintf .~puts "<option %s value='%s'>%s</option>\n"
         (if .~ord = tag then "selected" else "") tag text;
        .~(ord_options puts ord opts))>.
(* End of 'dir.meta' *)
?>
```