Ocsigen: Typing Web Interaction with Objective Caml

Vincent Balat

Laboratoire *Preuves, Programmes, Systèmes*
Université Paris 7
balat @ pps.jussieu.fr

Abstract

Ocsigen is a framework for programming highly dynamic web sites in Objective Caml. It allows to program sites as Ocaml applications and introduces new concepts to take into account the particularities of Web interaction, especially the management of URLs and sessions. This paper describes how Ocsigen uses the Objective Caml type system in a thoroughgoing way in order to produce valid XHTML and valid remote function calls through links and form clicking. It also describes how Ocsigen handles the progression of a Web user through a site, using sophisticated and high-level sessions mechanisms.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming languages*]: Language Constructs and Features

General Terms Languages, Design, Reliability

Keywords Objective Caml, Web Programming, Continuations

1. Introduction

Web programming has become a widespread activity and the languages and tools for programming dynamic Web sites are now numerous. Very often they are script languages (untyped or dynamically typed). They may be dedicated only to Web programming, (sometimes scripts embedded in a Web page, like PHP) or general purpose languages communicating with a Web server through CGI (Common gateway Interface), or with their own server (or a server extension). Most of these solutions, and even the most used of them, do not respond to the new requirements of modern Web sites, with very complex structures, sessions, and which provides a highly dynamic interaction with the user. A new challenge for Web programming is to manage to make Web sites behave more and more like applications, part of the interaction being computed on the server while the other takes place on the browser.

But we must keep in mind that there will always be a major difference between a Web site and a regular application, due to the way Web interaction occurs. The browsers give the users the ability to choose themselves their own routes in the site, to duplicate the threads of interaction, or to go back to a former state, and even resubmit a form. This is one of the main difficulties Web programmers have to cope with and the experience of many Web users shows that it is not always solved in the right way (see examples in [16]).

In this paper we will show a concrete way to take care of this issue, together with a strong use of types to ensure the correct functioning of the site, and to make its maintenance and future evolutions easier. It is implemented as a framework, called Ocsigen, containing a Web server written in Objective Caml (OCaml), with special features allowing to delegate the generation of pages to OCaml functions. All these features have benefitted from 15 months of experience and feedback from users that made them match more closely the concrete needs of Web programmers. Ocsigen is available online for download (see `http://www.ocsigen.org`), with a full tutorial. This paper describes version 0.4.0.

Overview of the paper

The first claim of this paper is that Web languages could take greater advantage of static typing. In that respect, our goal is twofold. First of all we want to ensure statically the correctness of generated pages with respect to standards defined by the W3C. Then, we want to consider dynamic pages as typed services with parameters and statically check that links and forms match the page they are pointing to.

The second claim on which our work relies is that functional programming fits perfectly well for Web programming and brings clever and unsuspected solutions to well known problems, in a very elegant and natural way. Christian Queinnec and Paul Graham have described independently how Web programming could take advantage of the notion of *continuation*, that can be manipulated easily using functional languages [24, 25, 14]. But there are still very few tools based on these ideas. This paper shows how they are implemented in Ocsigen and how we develop them to make them fit closer to the concrete needs of Web programmers.

Section 2 is devoted to an introduction to programming with Ocsigen. We show the types and syntax used for XHTML data, how works Ocsigen's web server, and how we can add new dynamic pages (services) to a site. Section 3 describes the techniques we use to type services, links and forms. Then, the different ways to manage the progress of a user through a Web site is described in section 4. In particular, we will show that Ocsigen allows to use two kinds of sessions.

2. Main principles

The Ocsigen Web server is implemented as an event-driven server, using cooperative light-weight threads implemented using monads by Jérôme Vouillon. The HTTP protocol has been implemented by Denis Berthod.

Copyright © 2006 ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *ML'06* September 16, 2006, Portland, Oregon, USA. (1-59593-483-9/06/0009)

Web sites are implemented as OCaml modules that are loaded dynamically while launching the server. In this section we will see how to write such modules. First, we will show how to construct and type XHTML pages (section 2.1), then we will discuss the techniques of Web programming with continuations (section 2.2), and show how you do this concretely and transparently with Ocsigen (section 2.3).

2.1 Typing XHTML

The web relies on norms designed by the *World Wide Web Consortium*. The format used for Web page publishing is the well-known HTML, based on SGML. It is being progressively replaced by XHTML, which is XML based. Even if web browsers interpret these standards in a rather loose way, the need to respect them is crucial, as they are the only way to ensure interoperability and accessibility. Today, very few sites are standard-compliant. Most Web programming tools generate HTML by simply printing text and the task to check validity is the responsibility of the conscientious programmer. But even by using validators, we cannot guarantee that all the generated pages of a dynamic site will always be valid, as we cannot foresee all possible outputs.

The solution to this problem consists in using well-suited type systems to check statically that the program cannot produce a wrong output. The Web page is generated as a tree data structure that is pretty-printed using the XML syntax. Some research works have developed languages with type systems devoted to XML manipulation (XDuce, CDuce [18, 1]) which would be a perfect solution to this problem. OCamlDuce [12], that we plan to use for an experiment in a future version of Ocsigen, is an extension of OCaml with these types. But for the time being, we decided to rely on the standard OCaml type system, that allows to type XHTML in rather good manner, as explained below.

Polymorphic phantoms

Like other functional languages, it is very easy with OCaml to manipulate tree data structures using variants (sum types). But the difficulty here is that each XHTML tag may occur in several contexts. For example a `<p>` tag may be the child of a `<body>` or a `<div>` tag (among many others), but the possible children for `<div>` and `<body>` are not the same. It makes the use of (non polymorphic) ML-style variants very difficult, as we may need to create several constructors for each tag, or introduce intermediate constructors that do not correspond to a tag. Fortunately, OCaml has another kind of variants, called *polymorphic variants*, that allow to use the same constructor for several types. In that respect, they are very close to XML types, and that's why we use them for typing XHTML.

In Ocsigen, polymorphic variants are used as *phantom types* [19]. That means that they occur only in type parameters and not in the data itself. Concretely, here is an excerpt of the definition of the type for XML trees:

```
type gen_elt =
  Element of string * attrib list * gen_elt list
| ...

type 'a elt = gen_elt
```

Thorsten Ohl's XHTML module

This type is abstract and we use functions with type constraints to construct the tree nodes. Figure 1 shows the types of some of these functions.

This technique is not new as we use an implementation of these functions due to Thorsten Ohl, that is also distributed as a distinct library. I am aware of another implementation very similar used by Alain Frisch for the Bedouin project [11].

Here is an example of a Web page created with this functions:

```
(html
  (head (title (pcdata "")) [])
  (body [h1 [pcdata "Hello"]]))
```

The `pcdata` function is used for raw text. Note that the `head` function takes as first parameter a `<title>` or `<base>` tag, as this is mandatory for XHTML, and then a list of optional tags.

An invalid use of these functions will lead to an error message while compiling. For example if we put a `<p>` tag instead of `<title>`:

```
(html
  (head (p [pcdata ""]) [])
  (body [h1 [pcdata "Hello"]]))

This expression has type [> `P ] XHTML.M.elt
but is here used with type
  [< `Base | `Title ] XHTML.M.elt
```

In XHTML, some tags cannot be empty. For example `<table>` must contains at least one row. To enforce this, the `table` function takes two parameters: the first one is the first row, the second one is a list containing all the other rows. More generally, mandatory tags are always given as separate parameters to the functions (see for example the `head` function on figure 1).

XHTML syntax extension

Ocsigen provides another way to construct Web pages, closer to the usual XHTML syntax. This techniques uses a Caml syntax extension that allows to write pages using a more familiar syntax, or to reuse XHTML code. The syntax extension is implemented using the Camlp4 preprocessor that comes with the standard OCaml distribution [21, 6]. Here is an example of an OCaml function returning a web page:

```
(fun s ->
  << <html>
       <head><title></title></head>
       <body><h1>$str:s$</h1></body>
     </html> >>)
```

The `$` character allows to include any OCaml expression inside XHTML code (antiquotations).

The syntax extension constructs OCaml code, using the same `'a elt` type in order to make possible to mix both syntax. It uses type annotations to set the type variable of each element and element list, and thus check validity.

Discussion about page generation

Using the techniques described above, Ocsigen allows to create pages that respect most of the validity constraints of XHTML. Note that the syntax extension is less strict than the first solution, because it does not check the presence of mandatory tags. Indeed, it does not use Ohl's functions but directly the `'a elt` type. That's why we recommend to use the syntax extension only to reuse XHTML code. As we will see later, very often you don't write large portions of XHTML when you program with Ocsigen, because the construction of Web sites is highly modular. Most of the time, you use higher level functions that create predefined portions of the page. That's why we think that the syntax with functions is totally acceptable.

The drawback of using polymorphic variants to type pages is obvious: you cannot produce other kinds of XML. That's why

```
type ('a, 'b, 'c) star = ?a:'a attrib list -> 'b elt list -> 'c elt
(* Star '*' denotes any number of children, including zero. *)

type common = [ `Class | `Id | `Title | `XML_lang ]

type block = [ `Address | `Blockquote | `Del | `Div | `Dl | `Fieldset | `Form
 | `H1 | `H2 | `H3 | `H4 | `H5 | `H6 | `Hr | `Ins | `Noscript | `Ol | `P | `Pre | `Script | `Table | `Ul ]

val html : ?a:[< `Version | `XML_lang | `XMLns ] attrib list ->
      [< `Head ] elt -> [< `Body | `Frameset ] elt -> [ `Html ] elt

val head : ?a:[< `Profile | `XML_lang ] attrib list ->
      [< `Base | `Title ] elt ->
      [< `Link | `Meta | `Object | `Script | `Style ] elt list ->
      [> `Head ] elt

val body : ([< common ], [< block ], [> `Body ]) star
```

Figure 1. Type of some XHTML generation functions

we consider switching to XDuce-like types in the future. Another advantage would be to be able to easily parse incoming XML data, for example to implement Web services or XForms-like form handling [26].

Elsman and Larsen developed a Web server in SML together with a module for valid XHTML generation [9, 8, 10]. They use an alternative way of checking types, based on a system of combinators with phantom types. It is stronger for checking validity but is more complex to use.

2.2 Programming the web in CPS

There are basically two ways to do functional Web programming. The first one consists in viewing the sending of a Web page by the server as a function call, (a question asked of the user) that will return for example the values of a form or a link pressed. The problem is that you never know in advance to which question the user is answering, because he may have pressed the back button of his browser (or he may have duplicated the page). Christian Queinnec solves this problem by using the Scheme control operator `call/cc` that allows to name the current continuation of the program and to go back to this continuation when needed.

The second solution is symmetric to the first one, as it consists in viewing the click on a link or form by the user as a remote function call. Each link or form of the page corresponds to a continuation, and the user chooses the continuation he wants by clicking on the page. This corresponds to a *Continuation Passing programming Style* (CPS), and has the advantage that it does not need control operators any more. Strangely, this style of programming, usually considered unnatural, is closer to what we are used to doing in traditional Web programming. Even in PHP, each script file, waiting on the server may be considered as a continuation waiting its parameters.

Like most of other continuation-based Web frameworks (but unlike PLT Scheme), Ocsigen uses the CPS solution as it seems to be a more natural way of programming. Furthermore the standard distribution of OCaml does not have control operators.

As an example, consider a web page that asks for a number from the user. The number is sent to the server, and then a second page is generated, proposing to enter a second number to be added to the first one. Then a third page is displayed, showing the result of the calculus. A first way of implementing this site consists in putting the first number in the second page, as a hidden form field or in the URL of the third page. Thus, both numbers will be transmitted to the third page. But what if the result depends on much more complicated data, that you can't put easily in a hidden field or in the URL? With continuations, the solution is very clear. When receiving the first parameter, the server creates dynamically a continuation corresponding to adding this number. Sending the second number will activate this continuation.

This view of a click as the choice of a continuation is a very big step towards a good understanding of Web interaction. In CPS style, continuations can be represented using closures, which is a strong argument in favour of functional Web programming.

However there are still some issues to solve and choices to make. The first one is: how and where to save continuations? There are two possible answers: either on the server or on the client, and both have advantages and disadvantages. Sending the continuation to the browser has the advantage that no space is needed on the server, making this solution more scalable. We need a marshaling technique, and an encryption method to ensure that no unsure code will be executed by the server. The main drawback is that it may sometimes entail the transfer of large amounts of data, from the server to the client and then back again.

The other solution is to keep the continuation on the server and to send only a tag (an integer) from which we can recover the continuation when a request comes from the browser. To save memory, continuations may be marshaled to disk by the server. This is the only solution implemented in Ocsigen at the moment (version 0.4.0), but we consider implementing both and let the user choose which one fits better his needs.

The second problem to solve is: how to attach a continuation to a precise URL? It is crucial to keep at least the main URLs of the site readable by the user, and easy to write, as they are entry points for the Web site. The programmer must have the possibility to choose himself the text of these entry points, and these texts must be fixed forever (because the user may bookmark the URL). Sections 2.3 and 4 will show how URLs are handled by Ocsigen to take these issues into account.

2.3 Creating services with Ocsigen

Let's summarise the main principles of Ocsigen seen above:

- First of all, each visit of a user on a web site must be considered as a fresh execution of an application,
- When clicking a link or sending form parameters, the user chooses the continuation of this program (CPS programming),

- All the continuations are kept on the server in a table. The key to find the right continuation is the URL and names of parameters sent to the page.

We call *entry point* to the Web site an URL together with the set of names of the parameters it takes.

Each Web site is an OCaml module (`.cmo` or `.cma`) or a set of modules, that are dynamically loaded when launching the server. A configuration file specifies which modules should be loaded, and, for modules defining Web sites, to which main URL they must be attached (thus you may have several Web sites attached to different URLs).

A major difference with traditional Web programming tools is that you don't have one file for each URL. All the URLs used by a Web site may be created from a single module. More precisely a module may create several entry points (as defined above). To each of them you must attach a default service (continuation), that we will call *"public service"*, as opposed to *"private services"* that are dynamically created for one particular user. The registration of public services in the table is done during the initialisation phase of the server (while loading the modules), using the function `register_new_service`. For example:

```
let hello = register_new_service
    ~url:["hello"]
    ~get_params:unit
    (fun _ () () ->
      (html
         (head (title (pcdata "")) [])
         (body [h1 [pcdata "Hello"]])))
```

The first parameter of this function (labelled `~url`) corresponds to the URL where the service will be available (here it may be something like `http://www.myserver.org/hello`). The second parameter (labelled `~get_params`) describes the parameters this service takes (here `unit` means none). The third one is the service attached to the entry point. All services are functions with three parameters: the first one allows the service to have access to server parameters (like the IP of the user or the user-agent of browser). The second one contains GET parameters (that is, parameters transmitted in the URL), and the third one is for POST parameters (transmitted by the browser in the body of the HTTP request). We will see in the following how to use them.

3. Typing Web interaction

3.1 Pages with GET parameters and Links

Let see an example of a service that takes GET parameters:

```
let hello_params = register_new_service
  ~url:["hello"]
  ~get_params:(int "i" ** string "s")
  (fun _ (i, s) () ->
    (html
      (head (title (pcdata "")) [])
      (body
        [p [pcdata "You␣sent:␣";
            strong [pcdata (string_of_int i)];
            pcdata "␣and␣";
            strong [pcdata s]]])))
```

Note that we use the same URL as before (`"hello"`). The two services will be discriminated using parameters names.

Here `int`, `string`, and `**` are functions to be used to describe the parameters a service takes (`**` is an infix function). The expression `(int "i" ** string "s")` means that the service is waiting for a pair of type `int * string` and that the integer corresponds to a parameter named `i` in the URL, and the string to a parameter named `s`.

The conversion from string to int is done automatically by the server, and an exception is raised when the conversion fails. It is possible to define custom base types for parameters, by implementing yourself the conversion functions to and from string. It allows to use sophisticated types (like "integers between 1 and 10" for example).

Now let's see how to make links to our registered services:

```
(fun sp () () ->
  (html
    (head (title (pcdata "")) [])
    (body
      [p
        [a hello sp.current_url [pcdata "click"] ();
         br ();
         a hello_params sp.current_url
                   [pcdata "click"] (42,"ocsigen")
        ]])))
```

Note that to create a relative link, you need to know the current URL, that can be found in the server parameters (first parameter of services). The `a` function, used to create links, takes four parameters. The first one is the service you want to link to, the second one is the current URL, the third one is the text of the link, and the last one corresponds to GET parameters you want to put in the link. In the first case, it is `()` as the service takes a unit parameter. In the second case, it is a pair `int * string`, and the link created will be something like `<a href="hello?i=42&s=ocsigen">click</a>`.

Note that the function `register_new_service` may be decomposed into `new_service` (creation of the entry point) and `register_service` (registration of a service for this entry point) in order to allow (mutually) recursive links. A public service must be registered on every entry point during the initialisation phase of the server, otherwise it won't start. This is to ensure that a link cannot lead to a service that does not exist. You cannot register a public service after the initialisation phase, as it does not really make sense (the new service would be available only the time during which the server will be up). Such dynamic verifications may be avoided by using a dedicated syntax for public service registration. We may implement a syntax extension to do this. It may also be possible to serialise the table of public services during a post-compilation phase.

Now we can already see the structure of a typical Ocsigen Web site: First of all you define all the entry points you'll need, then you register services for all these entry points. It is recommended to keep all the public registrations in a single `.ml` file. The services can be defined in other modules that are loaded before. This structuring of sites allows to program in a very modular (horizontal) way that contrast with the traditional (vertical) page-based programming.

3.2 Typing services

As we've seen in previous examples, the type of the last parameter of `register_new_service` depends on the `~get_params` parameter. Similarly, the last parameter of the `a` function for creating a link depends on the service it points to. This kind of behaviour is not easy to obtain, because the type of an OCaml function cannot depend on the value of its parameters.

You have the same problem if you want to implement a C-like `printf` function whose type depends on a format given as

parameter. We have successively implemented two ways to solve this problem, one using what we will call *format combinators* (also referred to as *functional unparsing*), and one using *generalised algebraic data types*.

Format combinators

Until version 0.3.x, Ocsigen used a programming technique that has been described by Olivier Danvy [4] and used for example for Type Directed Partial Evaluation [3, 5, 29].

The principle is the following: instead of representing types with variants (OCaml constructors), you use functions that, when combined together, will create all the functions you need that depends on this type. In the case of `printf`, it is easy, and the best way to understand is to look at its implementation:

```
type ('a, 'b) t = 'a -> 'b

let int : ('a, int -> 'a) t =
  fun k x -> (print_int x; k)

let string : ('a, string -> 'a) t =
  fun k x -> (print_string x; k)

let ( ** ) (f1 : ('b,'c) t) (f2 : ('a, 'b) t)
: ('a, 'c) t =
    fun k -> f1 (f2 k)

let printf (format : (unit, 'a) t) : 'a =
  format ()

# printf int 5;;
5- : unit = ()
# printf (int ** string ** int) 5 "hello" 3;;
5hello3- : unit = ()
# printf (int ** string ** int ** int) 5 "hello";;
5hello- : int -> int -> unit = <fun>
```

The machinery that actually prints the result is not in `printf` but in the format parameters.

In the case of Ocsigen, this is much more complicated, because the type is used in many situations: service registration, creation of the URL of a link or a form (or all other places where URLs may be used), dynamic typing of (GET or POST) page parameters, typing of a form w.r.t. the service it points to (see later), etc. All this leads to very complex programming, and has a significant drawback for the user: the type of the services has numerous parameters that makes error messages very difficult to read. For example here is the type of a service taking an integer and two strings (with Ocsigen 0.3.27):

```
(int name ->
   string name ->
     string name -> form_content elt list,
 form_content elt list,
 int -> string -> string -> [> `A ] elt,
 int -> string -> string -> [> `Form ] elt,
 int -> string -> string -> uri,
 int -> string -> string -> page, page, page,
 [ `Internal_Service of [ `Public_Service ] ])
  service
```

As we will se later, `'a name` is the type of names of parameters (actually `'a name = string`).

Generalised Algebraic Data Types

These difficulties convinced us to reimplement the typing of services using generalised algebraic data types. These types generalise ordinary algebraic data types by allowing to give the type signatures of constructors explicitly. This allows to define a parametric type, whose parameter depends on the constructor. Thus you can define for example the type of types, such that `Int i` be of type `int ty`, `Prod (Int i, Int j)` of type `(int * int) ty`, etc.

Generalised algebraic data types have been introduced by Xi et al [28] and may be implemented in OCaml in the future [23]. While waiting, we implemented this part using internally unsafe (and undocumented) OCaml features that break the typing system (but the use of the functions remains type-safe for the programmer of Ocsigen modules).

The type for a service taking one integer and two strings is now the following:

```
(int * (string * string),
 unit,
 [ `Internal_Service of [ `Public_Service ] ],
 [ `WithoutSuffix ],
 int name * (string name * string name),
 unit name)
  service
```

The first parameter of the type corresponds to GET parameters, and the second one to POST parameters. The third and fourth ones describe the kind of service, and the two last are used for typing forms (see later).

A third solution would have been to use dynamic types (if the languages had this feature). The advantage is that we could have used type inference to find the type of services! No more type information would have been needed! But this solution does not allow to use custom types like “integers from 1 to 10”.

3.3 Typing forms

In previous sections, we saw how we can give parameters to a link and ensure that GET parameters of a link (or POST form) will correspond to the service it points to.

Very often, parameters are sent by forms. We would like to guarantee that a form corresponds to the service it points to. More precisely, we need to ensure that:

- All parameters expected by the service are present,
- Types of parameters are correct,
- Names of parameters are correct.

In Ocsigen, the creation of a form is very similar to a link. Here is an example of a form (using GET method) towards our `hello_params` service:

```
get_form hello_params sp.current_url create_form
```

The only difference is that the content of the form is constructed by a function that takes the names of parameters as arguments. In the example, `create_form` is define by:

```
let create_form (intname,stringname) =
    [p [pcdata "Write␣an␣int:␣";
        int_input intname;
        pcdata "Write␣a␣string:␣";
        string_input stringname;
        submit_input "Click"]])
```

Thus we are sure to use the right names for parameters. The use of an abstract type `'a name` for parameters names instead of `string` allows to check the types of parameters, with dedicated widgets creators like:

```
int_input : int name -> [> `Input ] elt
```
or:
```
checkbox_input : bool name -> [> `Input ] elt
```

The only thing we can't verify by this method is the presence of all parameters needed by the service (and that they are present only once). That does not seem easy to do without using a dedicated type system for forms or by using a monadic constructs for web pages creation (as in WASH/CGI [27]). Such solutions will be explored in the future.

We are currently working on the implementation of more sophisticated types for services, like lists or option types. Some experimentations are available online in the last versions.

4. Managing Web sessions

4.1 URL as public entry points

Bookmarks

With Ocsigen, URLs are seen as public entry points to the site. A public service must be registered on each created entry point. This forces a link or form created with Ocsigen to always point to an existing service.

But the user may arrive to a service by two other methods: either by writing himself the text of the URL or by setting a bookmark in his browser. Ocsigen allows you to choose precisely what URL you want for each service, thus making easy to write manually the URL. The case of bookmarks deserves to be discussed further, especially when there are POST parameters.

POST parameters

Protocols allow two ways of transmitting parameters to a Web page. The first one, call "GET method" is to put the parameters encoded in the URL. The second one is called "POST method". With this submission method, parameters are not sent in the URL by the browser, but in the core of the HTTP request. These two methods are not equivalent, and a Web programmer must be very scrupulous when using one or the other. GET method allows to transmit non confidential parameters (even with HTTPS) that we want the user be able to memorise in a bookmark. POST method is to be used for all pages that will execute an action on the server (modification of a database, online purchase ...), or for all parameters that you don't want to remember in bookmarks (for example session parameters). While designing a Web programming tool you need to take both methods into account.

Consider the case of forms using the POST method. Imagine that the user submits a POST form and then bookmarks the resulting page. POST parameters won't be registered by the browser in the bookmark. When the user comes back using his bookmark, the service will be called without POST parameters.

Using traditional page based Web programming, you need for every page to verify the presence or not of parameters. This encourages you to anticipate all possible cases and write a default page when parameters are missing.

With Ocsigen, the presence of parameters is checked automatically by the server. To make the site more robust, and to allow the user to put bookmarks on services with POST parameters, we need to force the programmer to plan a fallback for each service with POST parameters. That's why the handling of POST parameters is different from GET parameters: *services with POST parameters must be registered on top of the same services without POST parameters*.

Services with POST parameters with Ocsigen

Here is an example of definition of a service with POST parameters, on top of our `hello` service.

```
let hello_with_post_params =
  register_new_post_service
    ~fallback:hello
    ~post_params:(string "value")
    (fun _ () value ->
      (html
         (head (title (pcdata "")) [])
         (body [h1 [pcdata value]])))
```

It is possible to create a service with both GET and POST parameters. But Web standards do not allow to mix GET and POST parameters in the same form (which would be useful in many cases). For that reason, if you create a service with GET and POST parameters, only POST parameters will come from the form. You must specify GET parameters exactly as in links towards GET services.

4.2 Auxiliary services

Up to this point, we have described how to write very static Web sites, where services are defined once and for ever. But as explained in section 2.2, it is very useful to allow to dynamically define new services (new continuations) for handling precise (and often private) tasks. That is what we call *auxiliary services*.

Like services with POST parameters, and for the same reason, auxiliary services must be registered on top of already existing public services. They are distinguished from their fallback public services by a special parameter.

Actually, auxiliary services may be used to dynamically define new URLs dedicated to one precise user (see following section), but also to distinguish between two ways of going to a particular URL. For example, the following program defines two ways to reload the page, one that will increment a counter, the other not.

```
let publicserv = new_service ["counter"] unit ()

let auxserv =
  new_auxiliary_service ~fallback:publicserv

let _ =
 let c = ref 0 in
 let page sp () () =
  (html
   (head (title (pcdata "")) [])
   (body
     [p
       [pcdata "i␣is␣equal␣to␣";
        pcdata (string_of_int !c); br ();
        a publicserv sp.current_url
                      [pcdata "reload"] (); br ();
        a auxserv sp.current_url
                      [pcdata "incr␣i"] ()]]))
 in
 register_service publicserv page;
 register_service auxserv
   (fun sp () () -> c := !c + 1; page sp () ())
```

The value of the special parameter should always be transmitted using the POST method. Indeed, this value is randomly generated and we cannot ensure that it will always be the same and that this auxiliary service will always be available (especially when creating dynamically new auxiliary services as we will see in next section).

But sending with POST method is not always possible. A way to send POST parameters in a link is to generate a hidden form with

POST method and to make a click on the link submitting the form using Javascript functions. For forms using GET method, it is more difficult.

To solve the problem without forbidding GET forms towards auxiliary services, we allow the special parameter to be transmitted using the GET method, but it is processed differently from other parameters: if the parameter is present but no auxiliary service is registered with this parameter, the fallback is used. This is not 100 percent safe, as you may fall by misfortune on another auxiliary service ...

4.3 Cookie based Sessions

By allowing to create dynamically auxiliary services, you allow to program very dynamic Web sites. But for security reasons, it would be better if a user could not have access to services registered for another user. Ocsigen allows to do that using a second notion of session, based on cookies.

Each connection of a user on a site must be seen as a fresh execution of the program. It will create a cookie, and a new table of services specific for this user. Obviously this is actually done only when needed. The function `register_service_for_session` allows to put a service in this table. The cookie is manipulated automatically by the server.

Figure 2 shows the implementation of the adder described in the example of section 2.2. As in this example, most of the time you will register auxiliary services in the session table.

One may argue that sessions based on cookies won't work on browsers that do not accept cookies. PHP allows two kinds of sessions, one cookie-based, and the other that transmits a session parameter in all URLs. This second version would be easy to implement in Ocsigen but we haven't done so for the moment. The reason is that these two kinds of sessions have different behaviours. Opening a session based on cookies will have an effect on the past of the interaction with the user. Imagine the user duplicates the browser before opening a session on one of them. Then he clicks on a link of the other one. If the session is cookie-based, this click will go to a page inside the session. If not, it will remains outside the session. There can be only one cookie-based session at the same time on a same browser for a Web site.

Both types of session are not interchangeable. Sessions with cookies are considered safer because the session parameter is not shown in the URL.

Registering private services

Ocsigen also allows to register in the session table new versions of public services. These services are specific for a user, and will replace the public services only for him. They are called *private services*. This is very useful for many sites, for example when you want the user to be able to connect using a login and a password. The service receiving connection informations (as POST parameters) will register all the pages he wants in the session table as side effect. An example is given in figure 3.

As for services with POST parameters and auxiliary services, a private service may be registered only on top of a public service. Thus you cannot bookmark a page that won't exist after the end of the session. This forces you to register a public service as fallback for all your session pages (for example with a connection box).

4.4 Actions

Actions are like services, but they do not generate any page. They are used when you want an effect to take place on the server, without changing the current page. By default, the current page is redisplayed after the action has been executed.

As an example, consider a site with many URLs (say a forum). Each URL has a a connected version and a non connected version, and you want a connection box on each non connected page. But you don't want the connection box to change the URL, you just want to connect and stay on the same URL, in connected version. Without actions, you would need to create a version with POST parameter of each of your public services, which would be very heavy to program.

With actions you just need to define only one action for the connection.

Figure 4 shows how two program a forum with two pages. A first page thats displays the list of messages, with for each message a link towards a second page taking the message number as argument and displaying the message with the comments of the users.

Figure 5 shows the same example with a connection box on each page, implemented with actions.

5. Discussion and Conclusion

5.1 Related works

There are already some Web programming tools for Objective Caml or SML, all using a rather traditional approaches to Web programming (mod_caml, NetCGI, AS/XCaml, WDialog, SMLServer ...).

Even in other languages, continuation-based tools are not numerous with respect to the large number of Web programming framework. However a full comparison of all of them would require several columns of this paper. We should mention for example Seaside [7] in Smalltalk, PLT Scheme, or the ViaWeb e-commerce application [15].

Felleisen and others showed that programming Web servers using functional languages has a lot of advantages [17]. They developed a way to program Web sites in direct style in Scheme without having to restore the state of control at each request from the user. They use an automatic translation based on CPS-transform, λ-lifting and defunctionalisation to make it possible to reuse traditional page-based technologies like CGI [20]. They also introduced a formal model of interactive Web programs [16].

One closely related approach is Peter Thiemann's WASH/CGI [27], implementing a CGI based system for programming Web pages in Haskell. It uses monads to create HTML pages, which makes possible an original and interesting way of handling the typing of forms. WASH/CGI also has a session mechanism allowing to solve the weaknesses of CGI scripts.

We should also mention the work by Schwartzbach et all about Web programming with a language called Bigwig, and then in Java (JWig) [2]. The last one extends Java with a session mechanism and an XHTML-like syntax, and guarantees page conformance and parameter matching using a suite of program analyses at compile time.

Other recent projects aim at creating new languages for Web programming, using functional programming for Web interaction, namely Websicola and Links.

Ocsigen distinguishes by many aspects, among which an original use of OCaml types, the way it uses URLs as entry points for the Web site, its management of GET and POST parameters, its session model, etc. With respect to most of other projects, Ocsigen's approach has an advantage that it doesn't need to extend the language.

5.2 Progress report of the project

We tried to design a tool very close to the expectations of Web programmers, but using really advanced features based on functional programming. Since the beginning of the project, the concepts in-

```
let calc = new_service ~url:["calc"] ~get_params:unit ()

let calc_post = new_post_service ~fallback:calc ~post_params:(int "i")

let _ =
  let create_form is =
    (fun n ->
       [p [pcdata (is^" + ");
           int_input n;
           br ();
           submit_input "Sum"]])
  in
  register_service
    ~service:calc_post
    (fun sp () i ->
      let is = string_of_int i in
      let calc_result = register_new_post_auxiliary_service_for_session
          ~fallback:calc
          ~post_params:(int "j")
          (fun sp () j ->
            (html
               (head (title (pcdata "")) [])
               (body
                  [p [pcdata (is^" + "^(string_of_int j)^" = "^(string_of_int (i+j)))]])))
      in
      let f = post_form calc_result sp.current_url (create_form is) () in
      (html (head (title (pcdata "")) []) (body [f])))


let _ =
  let create_form n =
    [p [pcdata "Write a number: ";
        int_input n;
        br ();
        submit_input "Send"]])
  in
  register_service
    calc
    (fun sp () () ->
      let f = post_form calc_post sp.current_url create_form () in
      (html (head (title (pcdata "")) []) (body [f])))
```

Figure 2. Adder

troduced in Ocsigen have evolved a lot to better match the needs of this specific kind of programming. The experience of our first users has been invaluable and we trust that Ocsigen gets every day closer to its goal to make highly dynamic Web sites really easy to develop and maintain. Ocsigen is one of the most advanced projects in this style.

The implementation is already usable. It can be downloaded under a free software licence. Some useful features will be introduced soon. For example "disposable" services, that can be used only once (for example for an online purchase). We are currently working on an extension of forms creation, allowing to use more types for services parameters, like lists, option types, booleans or association tables.

On a technical point of view, Ocsigen's Web server has been running on our Web site for months without problems. Some work is currently done to support full HTTP/1.1 protocol, and to make programming with light-weight threads easier.

We are also working on two major enhancement of the system: one concerns the automatic generation of (well typed) Javascript code to be executed on the browser [22]. This extension will allow to program really dynamic Web sites, using a technology named "Ajax" [13], that use a lot of client-side computation ("rich clients"). Sites will become concurrent programs, all written in the same statically-typed language (Ocaml with syntax extension). The other enhancement concerns interaction with databases, where much remains to be done to integrate a type safe query language.

5.3 Conclusion

We have shown a new way of programming dynamic Web sites based on functional programming and static typing, much more robust than traditional Web programming. We demonstrated that it is possible to do it without creating a new language, using advanced features of the Objective Caml language.

When trying to make application-like programming for Web sites, the main difficulty encountered is due to the fact that we depend on existing protocols and standards, and to the way browsers work. For example we must conform to the model of requests based on URLs. This dependency also makes the creation and typing of forms difficult. A new standard, called XForms [26] has been proposed by the W3C to replace XHTML forms (it is not implemented

```
let public_session_without_post_params = new_service ~url:["session"] ~get_params:unit ()

let public_session_with_post_params =
  new_post_service ~fallback:public_session_without_post_params ~post_params:(string "login")

let home sp () () =
  let f = post_form public_session_with_post_params sp.current_url
    (fun login -> [p [pcdata "login:␣"; string_input login]]) () in
  (html
     (head (title (pcdata "")) [])
     (body [f]))


let _ = register_service ~service:public_session_without_post_params home

let rec launch_session sp () login =
  let close = register_new_auxiliary_service_for_session
    ~fallback:public_session_without_post_params
    (fun sp () () -> close_session (); home sp () ())
  in
  let new_main_page sp () () =
    (html
       (head (title (pcdata "")) [])
       (body [p [pcdata "Welcome␣"; pcdata login; pcdata "!"; br ();
                 a hello sp.current_url [pcdata "new␣hello"] (); br ();
                 a close sp.current_url [pcdata "close␣session"] ()]]))
  in
  register_service_for_session ~service:public_session_without_post_params new_main_page;
  register_service_for_session ~service:hello
    (fun _ () () ->
      (html
         (head (title (pcdata "")) [])
         (body [p [pcdata "Hello␣"; pcdata login; pcdata "!"]])));
  new_main_page sp () ()

let _ = register_service ~service:public_session_with_post_params launch_session
```

Figure 3. Session with login

```
(* All the public services *)
let main_page = new_service ~url:[""] ~get_params:unit ()
let news_page = new_service ~url:["msg"] ~get_params:(int "num") ()

(* Construction of pages *)
let print_main_page sp () () = ... (* Display home page with login box *)
let print_news_page sp num () = ... (* Display message with login box *)

(* Services registration *)
let _ = register_service main_page print_main_page
let _ = register_service news_page print_news_page
```

Figure 4. A forum

```
(* All the public services and actions *)
let main_page = new_service ~url:[""] ~get_params:unit ()
let news_page = new_service ~url:["msg"] ~get_params:(int "num") ()
let connect_action = new_action ~post_params:(string "login" ** string "password")

(* Construction of pages *)
let print_main_page sp () () = ... (* Display home page with login box *)
let print_news_page sp num () = ... (* Display message with login box *)
let user_main_page user sp () () = ... (* Display home page for the user *)
let user_news_page user sp num () = ... (* Display message for the user *)

(* Services registration *)
let _ = register_service main_page print_main_page
let _ = register_service news_page print_news_page

let launch_session user =
  register_service_for_session main_page (user_main_page user);
  register_service_for_session news_page (user_news_page user)

let _ =
  register_action ~action:connect_action (fun login password -> launch_session (connect login password))
```

Figure 5. A forum with a connection box on each page

in browsers for the while). It may solve some typing problems by using an XML format to transmit parameters, but it does not solve the problem of sending both GET and POST parameters through the same form.

Ocsigen produces pages that respect most of the validity constraints of XHTML, and gives the insurance that URLs won't be broken, and that forms will lead to services that can handle them (with the limitations seen above). One of the strongest features of Ocsigen is the way it handles sessions in a transparent way, contrasting with traditional tools that force programmers to save and restore control state at each step of the interaction with the user.

Examples of figures 4 and 5 show that describing the functioning scheme of a Web site with Ocsigen, even with sessions is really easy. A module for a complete Web site is very concise. In the examples, all the parts that have been omitted are the code displaying the pages, that is usually delegated to functions from other OCaml modules. This code is also usually very short, as the use of OCaml functions allows to reuse a lot of code. Thus, you program Web sites in an highly modular way. Note also that these functions will not have to worry about presence of parameters nor about sessions handling. All this is done automatically by the server.

Acknowledgments

Many acknowledgements are due for all the people who took part in Ocsigen development (whose names are already quoted in the paper), for first users and for anonymous referees. Many thanks are due to Jean-Vincent Loddo, Jérôme Vouillon, Russ harmer, and all the people who participated in discussions about the project.

References

[1] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.

[3] O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*, pages 242–257, 1996.

[4] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[5] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Conference on Automata, Languages and Programming (ICALP)*, number 1443 in Lecture Notes in Computer Science, pages 908–917, 1998.

[6] Daniel de Rauglaudre. Camlp4 - reference manual.

[7] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside – a multiple control flow web application framework. In *Proceedings of ESUG Research Track 2004*, pages 231–257, 2004.

[8] Martin Elsman and Niels Hallenberg. *SMLserver—A Functional Approach to Web Publishing*, February 2002. (154 pages). Available via `http://www.smlserver.org`.

[9] Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.

[10] Martin Elsman and Ken Friis Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*. Springer-Verlag, June 2004.

[11] A. Frisch. The bedouin project, `http://sourceforge.net/projects/bedouin`.

[12] Alain Frisch. OCaml + XDuce. In Giuseppe Castagna and Mukund Raghavachari, editors, *PLAN-X*, pages 36–48. BRICS, Department of Computer Science, University of Aarhus, 2006.

[13] Jesse James Garrett. Ajax: A new approach to web applications.

[14] Paul Graham. Beating the averages `http://www.paulgraham.com/avg.html`.

[15] Paul Graham. Method for client-server communications through a minimal interface. United States Patent 6,205,469.

[16] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In *European Symposium on Programming (ESOP)*, April 2003.

[17] Paul T. Graunke, Shriram Krishnamurthi, Van der Hoeven, and

Matthias Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming (ESOP 2001)*, 2001.

[18] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[19] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*, pages 109–122, 1999.

[20] Jacob Matthews, Robert Bruce Findler, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring software for the web. *Journal of Automated Software Engineering*, 2004.

[21] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.

[22] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005. ACM Press.

[23] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 232–244, Charleston, South Carolina, January 2006.

[24] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP '2000 – International Conference on Functional Programming*, pages 23–33, Montreal (Canada), September 2000.

[25] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[26] W3C recommendation. Xforms 1.0.

[27] Peter Thiemann. Wash/cgi: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages (PADL'02)*, January 2002.

[28] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

[29] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300. ACM Press, 1998. Extended version available as the technical report BRICS RS-98-9.