Experience Report: Ocsigen, a Web Programming Framework

Vincent Balat   Jérôme Vouillon   Boris Yakobowski

Laboratoire Preuves, Programmes et Systèmes
Université Paris Diderot (Paris 7), CNRS
Paris, France
{vincent.balat, jerome.vouillon, boris.yakobowski}@pps.jussieu.fr

Abstract

The evolution of Web sites towards very dynamic applications makes it necessary to reconsider current Web programming technologies. We believe that Web development would benefit greatly from more abstract paradigms and that a more semantical approach would result in huge gains in expressiveness. In particular, functional programming provides a really elegant solution to some important Web interaction problems, but few frameworks take advantage of it.

The Ocsigen project is an attempt to provide global solutions to these needs. We present our experience in designing this general framework for Web programming, written in Objective Caml. It provides a fully featured Web server and a framework for programming Web applications, with the aim of improving expressiveness and safety. This is done by taking advantage of functional programming and static typing as much as possible.

Categories and Subject Descriptors D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; H.3.5 [*INFORMATION STORAGE AND RETRIEVAL*]: Online Information Services—Web-based services

General Terms Design, Languages, Reliability, Security

Keywords Ocsigen, Web, Networking, Programming, Implementation, Objective Caml, ML, Services, Typing, Xhtml

1. Introduction

In the last few years, the Web has evolved from a data-centric platform into a much more dynamic one. We tend now to speak more and more of *Web application*, rather than Web sites, which hints that the interaction between the user and the server is becoming much more complex than it used to be.

What is striking is that this evolution has not been induced, nor even followed, by a corresponding evolution of the underlying technology. The RFC specifying the version of the HTTP protocol currently in use dates back to 1999 and current HTML looks very much like the one we were using ten years ago. The main change is probably the increasing use of JavaScript, mainly due to implementation improvements, which made possible the advent of a new kind of Web applications.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08...$5.00

Web programming is highly constrained by technology (protocols, standards and browsers implementations). Commonly used Web programming tools remain very close to this technology. We believe that Web development would benefit a lot from higher level paradigms and that a more semantical approach would result in huge gains in expressiveness.

It is now widely known in our community that functional programming is a really elegant solution to some important Web interaction problems, as it offers a solution to the statelessness of the HTTP protocol (Queinnec 2000; Graham 2001; Hughes 2000). But this wisdom has not spread in the Web programming community. Almost no major Web framework is taking advantage of it.[1] We believe the reason is that functional programming has never been fully exploited and that one must be very careful about the way it is integrated in a complete framework in order to match precisely the needs of Web developers.

The Ocsigen project is trying to find global solutions to these needs. In this paper, we present our experience in designing Ocsigen, a general framework for Web programming in Objective Caml (Leroy et al. 2008). It provides a full featured Web server and a framework for programming Web applications, with the aim of improving expressiveness and safety. This is done by taking advantage of functional programming and static typing as much as possible (Balat 2006).

This paper is a wide and quick overview of our experience regarding this implementation. In section 2, we describe our use of functional programming. In section 3, we show how some very strong correctness properties can be encoded using Ocaml's type system. Finally, in section 4, we describe the implementation of a concrete Web application using our solutions.

2. Functional Programming for the Web

The Ocsigen project provides a Web server written in Objective Caml. This server offers all the features one would expect from a general purpose Web server, starting with a comprehensive support of the HTTP 1.1 protocol (Fielding et al. 1999) (including range requests and pipelining). Data compression, access control and authentication are all supported. The server is configured through flexible XML-based configuration files.

The server is designed in a modular way. It can therefore be extended very easily just by writing new modules in Ocaml. Among the modules currently available are a module for running CGI scripts, a reverse proxy (which makes it easy to use Ocsigen with another Web server), a filter to compress contents, *etc.*

In the remainder of this section, we highlight the concurrency model used for the Web server implementation, and then our main

[1] A notable exception being Seaside (Ducasse et al. 2004).

extension to the server, that is, Eliom, a framework for writing Web applications in Ocaml.

2.1 Cooperative Threads

A Web server is inherently a highly concurrent application. It must be able to handle simultaneously a large number of requests. Furthermore, composing a page may take some time, for instance when several database queries are involved. The server should not to be stalled in the meantime. We have chosen to use cooperative multi-threading to address this issue. Indeed, cooperative threads make it possible to write multi-threaded code while avoiding most race conditions, as context switches only occur at well-specified points. Between these points, all operations are executed atomically. In particular, it is easy to use safely shared mutable datastructures, such as hash tables, without using any lock.

We use the Lwt thread library (Vouillon 2008), which provides a monadic API for threads. With this library, a function creating a Web page asynchronously will typically have type:

$$\texttt{unit} \rightarrow \texttt{html Lwt.t}.$$

It returns immediately a *promise* (sometimes also called *future*) of type `html Lwt.t`, that is, a value that acts as a proxy for the value of type `html` eventually computed by the function. Promises are a *monad*. The `return` operator of the monad has type:

$$\texttt{'a} \rightarrow \texttt{'a Lwt.t}.$$

It creates an immediately fulfilled promise. The `bind` operator has type:

$$\texttt{'a Lwt.t} \rightarrow (\texttt{'a} \rightarrow \texttt{'b Lwt.t}) \rightarrow \texttt{'b Lwt.t}.$$

It takes as arguments a promise and a function to be applied to the value of the promise when it becomes available. The promise returned by the operator gives access to the value eventually computed by the function. This operator can be used for sequencing asynchronous operations, as well as for synchronization (for waiting for the completion of some operation before performing further operations).

In order to make it possible to use third-party non-cooperative libraries, Lwt also allows to detach some computations to preemptive threads.

2.2 Web Programming

Eliom (Balat 2007) is the most innovative part of the project. This Web server extension provides a high-level API for programming dynamic Web sites with Ocaml. Its design goals are twofold: to propose a new Web programming paradigm based on semantic concepts rather than relying on technical details, and to ensure the quality of Web application by using static typing as much as possible (this latter point is detailed in section 3).

The main principle on which Eliom is based is the use of first-class values for representing the *services* provided by the Web server. What we call a service is a proxy for a function which can be called remotely from a browser to generate a page or perform an action. Eliom keeps track of the mapping from URLs to services: instead of having one script or executable associated to each URL, like many traditional Web programming tools, Eliom's services are programmatically associated to URLs. This lets the programmer organize the code in the most convenient way. In particular, it makes it easy to share behaviors between several pages, as the same service can be associated to several URLs.

As an example, the following piece of code creates a service `srv` at URL `http://foo/bar` (on some server `foo`).

```
let srv = register_new_service
            ~path:["bar"]
            ~get_params:unit
            f
```

The first argument (labelled `~path`), corresponds to the path in the URL to which the service will be bound. The second argument (labelled `~get_params`) describes URL parameters (here none, as this service does not expect any parameter). The function `f` is used to produce the corresponding page.

Inside Eliom, one can generate an anchor linking to this service by applying the HTML anchor constructor `a` to the service `srv`, the current service context `sp` (which contains in particular the current URL, and is used to construct relative links), the text of the anchor `anchor_contents` and the unit value `()` corresponding to the absence of parameters.

```
a srv sp anchor_contents ()
```

Note that the service URL can be changed just by modifying the path at a single location in the source code, and all links will remain correct as they are computed automatically.

Several services can share the same URL, for instance when they expect different parameters. This means that a service will *not* respond if some of its arguments are missing or ill-typed, avoiding a whole class of hard-to-detect errors.[2] More generally, a full range of service kinds is provided, allowing to describe precisely how services are attached to URLs. This makes it possible to describe very flexible and precise Web interactions in just a few lines of code. A service can be associated to a path (and possibly parameters), to the value of a special parameter, or to both or them. The choice of the right service to invoke is performed automatically by Eliom.

Services can be dynamically created in response to previous interactions with the user. Their behavior may depend for instance on the contents of previous forms submitted by the user or the result of previous computations. This is implemented by recording the behavior associated to the service as a function closure in a table on the server. This is an instance of *continuation-based Web programming* (Queinnec 2000; Hughes 2000; Graham 2001), This is known to be a really clean solution to the so-called *back button problem*, but is provided by very few Web frameworks.

It is also possible to classify services with respect to the type of the value they return. Usually services return an HTML page, but it is also possible to build services sending for example files, redirections, or even no content at all. The latter are called *actions*, as they are used to perform an effect on the server (for example a database change). Eliom also provides a kind of action that will redisplay the page automatically after having performed the effect. It is also possible to write services that will choose dynamically the kind of output they want.

3. Typing a Web Application

3.1 XML Typing

Historically, browsers have treated HTML errors leniently. As a result, Web pages are often written in loosely standardized HTML dialects (so-called *tag soups*). However, the interpretation of malformed markup can vary markedly from one browser to the next. Ensuring that Web pages follow precisely existing specifications makes it more likely that they will be interpreted similarly by all Web browsers.

[2] This mainly concerns URLs written by hand or outdated: as explained in §3.2, Eliom automatically guarantees that links inside pages are correct.

Our framework ensures statically that all Web pages served by the server are well-formed and valid with respect to W3C recommendations. Two ways are provided to the developer to this end. The functor-based API of Eliom makes it possible to support these two choices.

The first way is to use the XHTML module developed by Thorsten Ohl.[3] This library provides combinators to write HTML pages. HTML element types are encoded using phantom types (Leijen and Meijer 1999) and polymorphic variants. The covariant abstract type of elements is type `'a elt`. For instance, the combinator `p` takes as argument a list of elements which are either text or inline elements, optionally some common attributes such as `class` or `id`, and returns a paragraph element:

```
val p : ?a:([< common ] attrib list) ->
        [< inline | ‘PCDATA ] elt list ->
        [> ‘P] elt
```

Here is a piece of code that builds a simple page, given a list of elements `the_page_contents`.

```
html
  (head (title (pcdata "Hello world!")) [])
  (body (h1 [pcdata "Hello world"]
         :: the_page_contents))
```

By using a syntax extension based on the Camlp4 preprocessor, HTML fragments can be directly incorporated into an Ocaml source code. The fragments are translated in Ocaml code that relies on the library above.

```
<< <html>
      <head> <title>Hello world!</title> </head>
      <body> <h1>Hello world</h1>
          $list:the_page_contents$ </body>
   </html> >>
```

The second way of writing valid Web pages is to use OcamlDuce (Frisch 2006), which brings together Ocaml and the CDuce language (Benzaken et al. 2003). The latter is specifically designed for XML, and allows to manipulate XML documents with very precise (in fact, exact) typing.

```
{{ <html>
      [<head>[<title>"Hello world!"]
       <body>[<h1>"Hello world"
              !{:the_page_contents:}]] }}
```

Unlike the XHTML library, which is specific to XHTML documents, OcamlDuce can be used to create any kind of XML documents, for instance, Atom feeds (Nottingham and Sayre 2005). The only drawback is that OcamlDuce is incompatible with Camlp4 at the moment, requiring somewhat complicated compilation schemes when OcamlDuce files are mixed with Ocaml files that use syntax extensions.

3.2 Typing Web Interactions

Eliom's first-class notion of service makes it possible to check the validity of links and forms. A service is represented as an abstract data structure containing all the information about its kind, its URL, its parameters, *etc.* As we saw in section 2.2, links are built automatically by a function taking as parameter a service, rather than a URL. This makes broken links impossible!

The types and names of service parameters are declared when constructing a service. Here is an example of service with two parameters `year` and `reverseorder`, of types `int` and `bool` respectively.

[3] `http://physik.uni-wuerzburg.de/~ohl/xhtml/`

```
let events_info =
  register_new_service
     ~path:["events"]
     ~get_params:(int "year" **
                  bool "reverseorder")
     (fun sp (year, reverseorder) () -> ...)
```

The third argument is the function implementing the service. It takes three parameters: `sp` corresponds to the current request context (it contains all information about the request, like the IP address of the client, its user-agent, etc). The second one is for URL parameters (GET) and the third one for the body of the HTTP request (POST parameters, here none).

When a request is received, the actual arguments are automatically type-checked and converted from string to the right ML datatype by the server. Note that the type of the function implementing a service depends on the *value* of the second parameter: here, it expects a pair of type `int * bool`. This is not easy to implement in Ocaml. We have considered two solutions to this problem. The first one, used before version 0.4.0 of Ocsigen was to rely on *functional unparsing* (Danvy 1998). The current solution consists in a simulation of *generalized algebraic datatypes* (GADT) (Xi et al. 2003; Pottier and Régis-Gianas 2006) implemented using unsafe features of Ocaml, anticipating their future introduction in the language. In the example above, `int`, `bool` and `**` are the GADT constructor functions, and the strings `"year"` and `"reverseorder"` are HTTP parameters names.

Parameters are statically checked when building a link with parameters. Concretely, the function `a`, that builds a link, takes as last parameter the arguments to be given to the service. These arguments will be encoded in the URL. Again, the type of this parameter depends on the service.

```
a events_info sp (pcdata "Last year seminars")
   (2008, true)
```

When generating a link, service parameter names are taken from the abstract structure representing the service. This ensures that they are always correct and makes it possible to change them without needing to update all links. Here, the generated relative link is:

```
events?year=2008&reverseorder=on
```

Eliom also provides some static guarantees that a form corresponds to its target service. As with link functions, the function that creates an HTML form takes as parameters the service and the information `sp` about the request. But instead of being directly given the contents of the form, it expects a function that will build the contents of the form. This function takes as parameters the *names* of the different fields of the form. The following example shows how to create a form to our `events_info` service:

```
get_form events_info sp make_form
```

Here is in an excerpt of the definition of the `make_form` function:

```
let make_form (year_name, reverseorder_name) =
  ...
  int_input ~input_type:‘Text ~name:year_name ();
  bool_checkbox ~name:reverseorder_name ();
  ...
```

The functions `int_input` and `bool_checkbox` are used to create respectively an "input" form widget, and a checkbox. To ensure a correct typing of the fields, we use an abstract parametric type `'a param_name` for names instead of simply type `string`. The parameter of this type is a phantom type corresponding to the type of the service parameter. Each function generating form widgets uses the appropriate type for the name of the parameter it corresponds to. For instance, the `~name` parameter of the function `int_input` above has type `int param_name`, whereas it has type `bool param_name` for the function `bool_checkbox`. This ensures that the field names correspond exactly to those expected by the service, and that their types are correct. But there is no guarantee that all required parameters are present, nor that the same parameter is not used several times in the form. Indeed, this would require a very sophisticated type system for forms, which would also need to interact gracefully with the type system for HTML. Rather than trying to write an hazardous and complex extension to Ocaml's type system, we decided to relax somewhat the static checks for forms.

3.3 Typing Database Accesses

Database accesses are crucial for Web programming. We have found it very convenient to use PG'OCaml[4], an interface to PostgreSQL[5] written by Richard Jones. In particular, Ocsimore, our content management system (see section 4), relies on it. We actually included some changes in PG'OCaml to turn its implementation into monadic style, in order to make it usable with Lwt. Thus, even though we do not use preemptive threads, queries still do not block the Web server.

The most noteworthy feature of PG'OCaml is that SQL statements are statically typed, with type inference. Another key point is that it is immune to SQL code injection vulnerabilities, as queries are automatically compiled into SQL prepared statements (which are pre-compiled, well-formed, queries that can receive arguments).

Static typing relies on the '`DESCRIBE statement`' command provided by recent versions of PostgreSQL. This command returns the types of the placeholders and return columns of the given statement. At compile time, SQL statements inside the Ocaml code are thus fed into the PostgreSQL frontend by a Camlp4-based preprocessor, which answers with their types. Types are then converted back into Ocaml types and used to generate the appropriate code. As an example, the following code

```
fun db wiki -> PGSQL(db)
  "SELECT id, contents FROM wikiboxes
   WHERE wiki_id = $wiki"
```

defines an Ocaml function of type

$$\texttt{db} \rightarrow \texttt{wiki\_id} \rightarrow (\texttt{wikibox\_id} \times \texttt{string option})\ \texttt{list}.$$

The field `contents` is an SQL text field which can be NULL. It is thus mapped to the Ocaml type `string option`.

The fact that queries are typed proved extremely useful, as it helps to find out rapidly which queries have to be modified whenever the database structure is changed during program development.

We see a few potential improvements to PG'OCaml. First, the code passed to PG'OCaml must be a valid SQL query. Hence, it is not possible to write queries as subblocks that are concatenated together, as is often done. It would be interesting to incorporate a comprehension-based query language (Trinder 1992) into Objective Caml. Second, SQL fields are often mapped to the same Ocaml type. In the example above, we have in fact `wiki_id = wikibox_id = int32`. Making `wiki_id` and `wikibox_id` abstract types in the Ocaml code results in a lot of (explicit) conversions, including problematic ones inside containers. We would like to use explicit coercions `(id :> int32)` and `(int32 :> id)` instead, but this is not currently possible in Ocaml.

[4] `http://developer.berlios.de/projects/pgocaml/`

[5] `http://www.postgresql.org/`

4. An Application: Writing a Wiki with Ocsigen

4.1 An Overview of Ocsimore

We have started the development of Ocsimore, a content management system written using Eliom. At the moment, it mostly consists in a wiki, with an advanced gestion of users and groups. Currently in final beta state, it is already used to publish the PPS laboratory website (`http://www.pps.jussieu.fr/`).

At the heart of Ocsimore is the notion of *box*. Wiki pages are composed by putting together or nesting boxes. This provides strong possibilities of modularity and code reuse: HTML code can be shared between several pages simply by putting it in a common box. Moreover, a box can be a *container*, that is, it can contain a hole that is to be filled by the contents provided by an outer box. In the example below, the first box has a hole (named by convention `<<content>>`) and is included in the second box.

```
Box 1:
  The text at the end is in bold: **<<content>>**.
Box 2:
  Let us call 1: <<wikibox box='1' | In bold >>
```

The second box is thus displayed as:

"Let us call 1: The text at the end is in bold: **In bold**."

The default wiki syntax of Ocsimore is Wikicreole (Sauer et al. 2007). It is translated into well-formed XHTML using OcamlDuce. Ocsimore features an authentication mechanism, for either PAM, NIS or Ocsimore-specific users. Wikis and wikipages can be assigned read, write and administration rights. The permissions system is very general, with the possibility to put users into groups which can themselves be nested. Moreover, Ocsimore features a notion parameterized groups. For example, the groups `WikiWriters` and `WikiReader` are parameterized by the id of a wiki, and `WikiWriters(3)` is the group of the users that can write in the third wiki. One can also add generic inclusions between parameterized groups with similar arguments, and `WikiWriters(`i`)` is included in `WikiReaders(`i`)` for any i.

4.2 General Structure of the Code

Ocsimore is written using the object system of Ocaml in order to define modifiable and extensible widgets in a modular way. This makes it easy to add extensions without any modification to the core system. For instance, a comment system (forum, blogs, news) is currently being implemented. The wiki is also extensible: syntax extensions can be implemented and then included inside wiki boxes.

Ocsimore makes use of Eliom's most advanced features. For instance, it takes advantage of *non-attached services*, *i.e.* services that are not associated to any path (they are implemented using special URL parameters). These services are used for implementing a connection widget on each page of the site in just a few lines of code. Indeed, we do not have to consider that each page may optionally take credential information as parameters. Instead, the special connection service just performs the action of opening a session and then triggers the redisplay of the current page after logging.

The same kind of service is also used for editing wiki boxes. As the central notion in our wiki is not the notion of page but the notion of box (where a box can be included in several pages) it is important to keep the information of the current page while editing a box. This behavior is really easy to implement using Eliom, and is a good example of the simplification of the code induced by Eliom's high level features.

It is noteworthy that making the implementation of common complex Web interactions so easy has an impact on the ergonomics of Web sites. The example of the connection box is eloquent: in many sites, lazy PHP programmers prefer having the connection box only in the main page of the site, rather than duplicating the code for each page.

5. Conclusion

A few other projects also take advantage of functional programming for the Web. The two most closely related are Links (Cooper et al. 2006) and Hop (Serrano et al. 2006). A few other tools have also implemented continuation-based Web programming: Seaside (Ducasse et al. 2004), Wash/CGI (Thiemann 2002) and PLT Scheme (Krishnamurthi et al. 2007). Eliom departs from all these others projects in that it is based on Ocaml and proposes a very rich set of services.

The wide overview of the Ocsigen project we have given in this paper demonstrates that building a Web programming framework is complex, as many very different issues have to be addressed. Continuation-based Web programming is a key notion of the system but is not sufficient in itself. It needs to be integrated within a full environment. We believe that, far beyond this pecular point, functional programming is the ideal setting for defining a more high-level way of programming the Web. It allows the programmer to concentrate on semantics rather than on implementation details. Note that this abstraction from low-level technologies does not entail any limitation but offers a huge step forward in expressiveness.

One of the main concerns of our project has been to improve the reliability of Web applications using static typing, which is at the opposite of traditional Web programming, based on scripting languages. We think this evolution is necessary because of the growing complexity of Web applications. Our experience in writing application with Eliom and in implementing the whole system itself shows that relying heavily on sophisticated features of the typing system simplifies a lot the maintenance and evolution of large pieces of software.

For all this project, we made the choice of using the Ocaml language rather than defining a new one. This makes it possible to take full advantage of the large set of available Ocaml libraries. We were surprised of being able to encode most of the properties we wanted using Ocaml's type system. Very few things are missing (a better typing of forms is one of them).

Up to now, we have concentrated mainly on server-side programming. We intend to extend this work to other aspects of Web programming, namely database interaction and client-side programming. This last point is really challenging as it is not obvious how to build a Web site where some parts of the code run on the server and other parts on the client, with strong safety guarantees. Our first experiment in that direction has been the implementation of a virtual machine for Ocaml in Javascript (Canou et al. 2008).

Currently, the Ocsigen implementation is mature enough to be used for developing and operating real Web sites. Ocsigen is an open source project with a growing community of users, who have already developed significant Eliom-based applications, and who were a great help in building a strong and usable tool. We thank them all.

References

Vincent Balat. Ocsigen: Typing Web interaction with Objective Caml. In *International Workshop on ML*, pages 84–94. ACM Press, 2006. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159889.

Vincent Balat. Eliom programmer's guide. Technical report, Laboratoire PPS, CNRS, université Paris-Diderot, 2007. URL `http://ocsigen.org/eliom`.

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. ISBN 1-58113-756-7.

Benjamin Canou, Vincent Balat, and Emmanuel Chailloux. O'browser: Objective Caml on browsers. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 69–78, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-062-3. doi: http://doi.acm.org/10.1145/1411304.1411315.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO 2006*, 2006.

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

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.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`.

Alain Frisch. OCaml + XDuce. In *International conference on Functional programming (ICFP)*, pages 192–200, New York, NY, USA, 2006. ACM. doi: http://doi.acm.org/10.1145/1160074.1159829.

Paul Graham. Beating the averages, 2001. URL `http://www.paulgraham.com/avg.html`.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay Mccarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme Web server. In *Higher-Order and Symbolic Computation*, 2007.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*, pages 109–122, 1999. URL `citeseer.ist.psu.edu/leijen99domain.html`.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Jérôme Vouillon, and Dider Rémy. The Objective Caml system. Software and documentation available on the Web, 2008. URL `http://caml.inria.fr/`.

Mark Nottingham and Robert Sayre. The Atom Syndication Format. RFC 4287, December 2005.

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.

Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *International conference on Functional programming (ICFP)*, pages 23–33, Montreal (Canada), September 2000.

Christoph Sauer, Chuck Smith, and Tomas Benz. Wikicreole: a common wiki markup. In *WikiSym '07: Proceedings of the 2007 international symposium on Wikis*, pages 131–142, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-861-9. doi: http://doi.acm.org/10.1145/1296951.1296966.

Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the Web 2.0. In *Dynamic Languages Symposium*, October 2006.

Peter Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *PADL '02*, January 2002.

Phil Trinder. Comprehensions, a query notation for DBPLs. In *DBPL3: Proceedings of the third international workshop on Database programming languages : bulk types & persistent data*, pages 55–68, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1-55860-242-9.

Jérôme Vouillon. Lwt: a cooperative thread library. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-062-3. doi: http://doi.acm.org/10.1145/1411304.1411307.

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.