O'Browser: Objective Caml on browsers

Benjamin Canou[1,2, *]
[1] Laboratoire d'Informatique de Paris 6
LIP6 - CNRS UMR 7606
Université Pierre et Marie Curie – Paris 6
Case 169, 75252 Paris Cedex 05, France
Benjamin.Canou@lip6.fr

Vincent Balat[2, **]
[2] Preuves, Programmes et Systèmes
PPS - CNRS UMR 7126
Université Paris Diderot – Paris 7
Case 7014, 75205 Paris Cedex 13, France
Vincent.Balat@pps.jussieu.fr

Emmanuel Chailloux[1]
[1] Laboratoire d'Informatique de Paris 6
LIP6 - CNRS UMR 7606
Université Pierre et Marie Curie – Paris 6
Case 169, 75252 Paris Cedex 05, France
Emmanuel.Chailloux@lip6.fr

Abstract

We present a way to run Objective Caml programs on a standard, unmodified web browser, with a compatible data representation and execution model, including concurrency. To achieve this, we designed a byte-code interpreter in JavaScript, as well as an implementation of the run-time library. Since the Web browser does not provide the same interaction mechanisms as a typical Objective Caml environment, we provide an add-on to the standard library, enabling interaction with the Web page. As a result, one can now build the client side of a web application with the standard Objective Caml compiler and run it on any modern web browser.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Run-time environments; H.5.3 [*Information interfaces and presentation*]: Web-based interaction

General Terms Languages, Design, Experimentation

Keywords Virtual machine, Web browsers, Objective Caml, Document Object Model, JavaScript

A preview of O'Browser is available for testing at (1)[1], including a tutorial scripted in Objective Caml.

1. Introduction

From its creation and up to a few years ago, one could define the World Wide Web as a system of interlinked hypertext documents accessed via the Internet. But that definition, still present in many dictionaries, is now outdated. It is not possible any more to speak about the Web without mentioning dynamism. That evolution happened in two steps. The first one was "server side" dynamism. It has been a small revolution of the nature of the Web itself, as it destroyed a strong invariant, namely the uniqueness of the association between URLs and documents. The second one, dynamism on client side, is currently taking place and the technologies involved are so unsatisfactory, that it is far from being completed.

At the beginning, client side dynamism was limited to small scripts, performing basic actions on pages, but such pages are evolving towards real applications running in browsers, or even distributed between a browser and one or several servers. Implementing that kind of application requires taking into account new issues, like communication problems (that will not be addressed in this paper) or portability. As we have very little influence on what succeeds on client side (browsers), we must depend on standards, and, more regrettably, on implementations. Unfortunately, most of the time, Web technologies have not been designed with the current evolution of the Web in mind and solutions need to take into account these constraints, at least for browser side.

Despite these problems, several recent projects aimed at proposing cleaner integrated solutions for the new Web. The first point is to use compiled and statically typed languages instead of traditional Web scripting languages. Besides the advantage in terms of efficiency, it allows to check statically the programs. Sophisticated uses of type systems have been proposed to check the consistency of Web interaction (Balat 2006; Cooper et al. 2006; Balat 2007), and to ensure that pages will respect standards (Benzaken et al. 2003; Hosoya and Pierce 2003).

All these projects also propose to forget the old "page-based" view of the Web that does not fit well the new kind of applications we want to build, and especially the notion of session.

The third common point of these projects is to use the same languages for programming all three tiers of a Web application (Leroy and Loddo 2003; Cooper et al. 2006; Serrano 2007), avoiding the need to resort to different languages for server side, client side and database access, and thus facilitating communication, compatibility, mobility, etc. This paper takes place in the Ocsigen project (2), initiated in 2004 to explore new techniques for programming complete Web applications, taking advantage of the functional programming paradigm and static typing. It distinguishes itself by not creating its own language, but aiming at allowing to use the full power of Objective Caml (Leroy et al. 2007) for programming the Web. In that perspective, we describe here an experiment on how to program the client side using Objective Caml.

Using the same language for server and client sides requires generating the code to be executed on the browser, using a technology it can understand. This also has the advantage of unburdening the Web programmer of the problems of compatibility between browsers, and may even allow to target several execution platforms.

There are currently mainly two ways to achieve this: either use a virtual machine implemented on the browser (often in a plug-in) or compiling a language towards JavaScript (Flanagan 1998). In

* Work partially supported by the Foundation *Digiteo-Triangle de la Physique*.
** Work partially supported by the *Agence Nationale pour la Recherche*.

[1] We use the notation (n) to reference external links; see section Links at the end of the document for full URLs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ML'08, September 21, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-062-3/08/09…$5.00

this paper, we propose a third way: an implementation of a virtual machine in JavaScript for an existing language (here Objective Caml). It allows running, on a browser, Objective Caml programs that have been compiled using the standard Objective Caml byte-code compiler (without any modification). To make this possible, we also reimplemented in JavaScript the Objective Caml run-time library and a new standard library allowing the interactions with the Web page. Thus, it is possible to use it for real client side dynamic Web programming, in "Web 2.0" style, fully integrated to the Web page (unlike some plug-ins that use their own displaying system in a box within the Web page). For example, you can perform dynamic changes of the XHTML code from your Objective Caml program.

With respect to the use of a plug-in, and like compiling towards JavaScript, it has the advantage that it requires no modification of the browser, and thus can be deployed easily. With respect to a compiler, we reached at very low cost the full power of Objective Caml (even preemptive threads). All current implementations of JavaScript are too slow to run really complex programs, so our implementation suffers from this limitation and enforces it with a factor of about ten in execution time. But we demonstrated by writing real examples that it is usable for all what is currently done in JavaScript in today's Web sites, and even more. The flexibility and programming comfort we gain foreshadows much more dynamic Web applications than what is done at the moment. O'Browser will fit perfectly in a complete Web programming framework, providing power and ease of use for the programmer and the user. We can consider using it as a flexible alternative to a run-time system implemented as a plug-in.

Section 2 is a short historical survey on the attempts to program web clients with ML dialects. Section 3 will then present our model, followed in 4 by some experimental results and examples. Finally, after a review of some related works in section 5, we shall discuss on this experiment, and present our future works in section 6.

2. ML languages and Web browsers

Functional languages, including the ML family, have followed the evolution of the client-server applications: from fat clients (large size applications with periodic connection to a central server) to thin clients (Web browsers where computations are on the server). And now to "Rich Internet Applications" (RIA), where the division of work is more distributed between servers and clients. To appreciate this evolution, we describe several experiments of RIA using one of the ML dialects, Objective Caml or Standard ML.

2.1 Applets

One of the first approaches to implement ML applets has been the MMM experiment (3), a Web navigator written in Objective Caml and running Applets in Objective Caml (Rouaix 1996). This work has proved that the Objective Caml language was a valuable choice to build a Web browser, mainly on separate compilation and security, type safety and isolated run-time environment. The security properties applets are formalized in (Leroy and Rouaix 1999).

Another attempt was to embed the Objective Caml top-level (4) inside a more commonly used browser as a plug-in for Mozilla or Firebird. These navigators provide a graphical area for the applets, as an X-window window, and so a graphical Objective Caml application can use this area to interact. A source program is downloaded, compiled to byte-code and then evaluated. A special top-level can be built with a limited library, without system or IO functions, to ensure security. There is no check of the downloaded source program since its successful compilation ensures that the

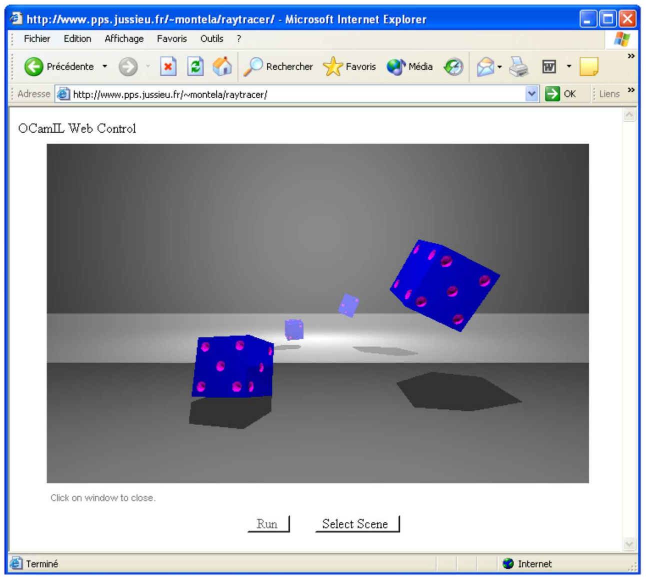

http://www.pps.jussieu.fr/~montela/raytracer/ - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente
Rechercher
Favoris
Média
Adresse http://www.pps.jussieu.fr/~montela/raytracer/
OK
Liens
OCamIL Web Control
Click on window to close.
Run
Select Scene
Terminé
Internet

Figure 1. A .Net applet running a ray-tracer in Objective Caml

produced byte-code can be executed.

Because the main Web browsers embed a Java Virtual Machine to run Java Applets, another way to create ML Applets is to use a compiler from ML to Java. We can cite MLj (5) which is a compiler from SML to Java byte-code. Two papers (Benton et al. 1998), (Benton and Kennedy 1999) describe the compiler architecture and its intermediate languages, and present benchmarks. Applets can be found on the MLj project page. On the Objective Caml side, the Ocaml-Java project (6) aims to provide interaction between Objective Caml and Java, including applets development.

The more recent platform .NET also allows to build applets. The interoperability must be with the CLR/C# model. For that, the MLj compiler has evolved to SML.NET (7) and the paper (Nick Benton and Russo 2004) describes this experience of interoperability in the .NET world.

The OCamil project (8) compiles the whole Objective Caml distribution, (including the top-level) to .NET managed code (Montelatici et al. 2005). Communications between Objective Caml and C# cannot be direct: C# and Objective Caml objects have to be interfaced. This is done with an IDL (*Interface Description Language*) and a code generator called O'Jacaré.net (Chailloux et al. 2004). Figure1 shows an adaptation of the ICFP Programming Contest 2000: a ray tracer program.

Thanks to all these experiments, the ML community is now able to build RIA as applets for the main Web browsers.

2.2 DOM

Another approach for rich client is to use the DOM (*Document Object Model*) (9) (DOM). The DOM is an interface allowing a program to modify dynamically the content or structure of a document which can be included inside a Web page. This interface is language independent. For example for Mozilla's browsers, the DOM implementation is based on the XPCOM components, which can be written in C, C++ or JavaScript. There are also some bindings between Java and XPCOM components as Java DOM API (10) or Java plug-in (11).

But the most famous way to manipulate the DOM is to script actions in JavaScript. In this case, JavaScript can be seen as a glue language for libraries, written as XPCOM components or directly in JavaScript. Among the most popular JavaScript libraries, we can cite jQuery or Scriptaculous (12).

The Google Web Toolkit (13) (GWT) provides a java2javascript translator to produce AJAX[2] applications. This method has been used to translate ML dialects as Objective Caml (ocamljs (14)), F# (Afax (15)) and Standard ML (smltojs (16)) programs to JavaScript. In section 4, we shall compare our solution to ocamljs and smltojs.

3. Description of our solution

Up to this point, the reader probably still has in mind the question *"Why do we want to run Objective Caml code in a browser, and why through JavaScript?"*. In this section, we first give our answer to this question and describe our experimental approach. We then present the API (*Application Programming Interface*) of O'Browser. Finally, we explain the execution model of O'Browser, with our answers to some selected technical problems which arose during its implementation.

3.1 Motivations

Why JavaScript? The answer to this part of the question is practical. Unlike technologies like Adobe Flash, Java or other plug-ins, JavaScript is available and enabled by default in all modern browsers. Furthermore, in some closed devices (like the Apple iPhone, the Nintendo Wii or some set top boxes), there is no way to install additional plug-ins. Hence, to be able to execute code in a Web page, one has to use the provided JavaScript engine. In other words, JavaScript, if neither the safest nor the fastest language, is the key to portability.

Why Objective Caml? Again, we wish to write a complete Web application in one language, in our case Objective Caml. Indeed, as shown in (Balat 2006), Objective Caml is a fine tool to develop the server part of a Web application. It is well adapted to tree manipulations, has a type system authorizing the static checking of many security issues (like form parameters' types), and is efficient enough to write a full-featured Web server. The maturity of the Objective Caml compilers gives a strong confidence in the safety of the execution, which is important in a hostile environment like the Web and would be hard to achieve if we were designing a new language and compiler from scratch. Moreover, using a well developed language gives access to numerous existing libraries, and, last but not least, is a good way to reach an audience since there already exists a quite active community around the language.

The next step in our effort to write Web applications in Objective Caml is therefore to run Objective Caml code on the client. A way to run Objective Caml programs through JavaScript is to write a compiler. However, since there already exists a byte-code compiler, another option is to write a virtual machine in JavaScript. The choice we made is obviously arguable, but here are some reasons behind it.

- As said earlier, we want to keep the gain of using a mature and well tested compiler.
- The Objective Caml byte-code format rarely changes from one version to another, which is not the case of the compiler, so it is easier to maintain.
- The abstraction we get from the JavaScript execution model enables us to get close to Objective Caml's semantics. For example, we provide preemptive threads, which do not exist in JavaScript, as will be detailed later in this section.
- It is well known that debugging JavaScript code is hard, in particular debugging generated code. Debugging a virtual machine is easier, by running it step by step and comparing its state to the original Objective Caml virtual machine.
- It sounded like fun and we did really want to see if it was feasible.

[2] AJAX (*Asynchronous JavaScript And XML*) is a commercial name for web applications heavily using the JavaScript possibility to perform HTTP requests

3.2 A brief chronology of the experiment

In our early experiments, we tried to mimic the standard execution environment within the browser, including a terminal emulator and the ⌜Graphics⌝[3] library to be able to run unmodified Objective Caml byte-code programs as shown in figure 2. We included a step-by-step debugger to help us design and optimize the core of the virtual machine.

As we were implementing the run-time library primitives, our scepticism on the viability of the approach faded. Indeed; the virtual machine was able to run, even if quite slowly, programs like a Mandelbrot set computation or the Knuth-Bendix rewriting algorithm (which is used in Objective Caml test suites because of its mixing of exceptions, string manipulations, functional style and allocations). In the end, we were close to being able to run the Objective Caml interactive top-level (the limitations came from the difficulty to mimic accesses to the file system which are needed by the Objective Caml compiler).

This led us to the conclusion that the approach of running Objective Caml programs on the client by using the standard Objective Caml compiler and a virtual machine in JavaScript is feasible.

However, if it was useful to develop the virtual machine core, our simulation of a standard Objective Caml environment was obviously not adapted to writing Web applications. So, the next step was to tweak the standard library to be able to manipulate the hosting Web page, and to remove input/output functions which make no sense in a Web browser environment.

The current version we present in this section includes this modified standard library and is simply a standalone *vm.js* file containing an ⌜exec_caml(url , argv)⌝ function taking an URL to a byte-code file as argument, to be used as shown in listing 1. The virtual machine implementation is fully re-entrant so several Objective Caml programs can run at the same time in one page.

```
// waits for the page to be loaded
window.onload = function () {
  exec_caml ("client_side.exe.uue");
  // since JavaScript supports 7bit ascii ,
  // we UUencode bytecode files
}
```

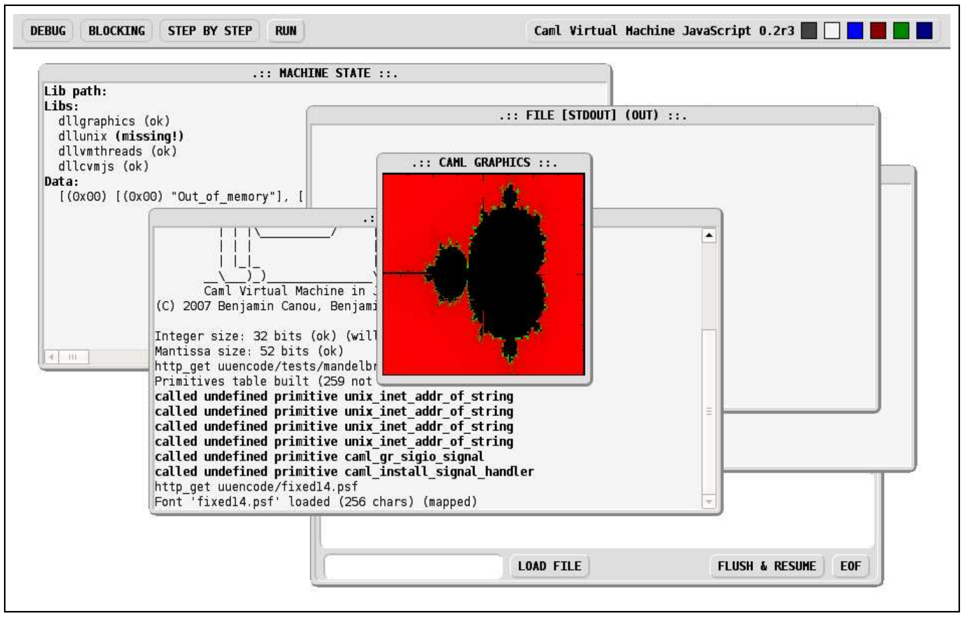
Listing 1. The only JavaScript code remaining: run ocaml

3.3 The modified standard library

As said in section 3.2, our first environment simulated a standard Objective Caml environment in which input/output functions were redirected to terminal emulators. In particular, it was possible to write page elements in the virtual console by printing XHTML code. However, as explained in section 2, a cleaner and more modern way to create page element is to use the DOM (*Document Object Model*).

The DOM If the reader is not familiar with the DOM, here is a quick and simplified description from a JavaScript point of view.

[3] We use the notation ⌜code⌝ for code extracts in the text



```
Lib path:
Libs:
  dllgraphics (ok)
  dllunix (missing!)
  dllvmthreads (ok)
  dllcvmjs (ok)
Data:
  [(0x00) [(0x00) "Out_of_memory"], [
```

```
     Caml Virtual Machine in J
(C) 2007 Benjamin Canou, Benjami

Integer size: 32 bits (ok) (will
Mantissa size: 52 bits (ok)
http_get uuencode/tests/mandelbr
Primitives table built (259 not
called undefined primitive unix_inet_addr_of_string
called undefined primitive unix_inet_addr_of_string
called undefined primitive unix_inet_addr_of_string
called undefined primitive unix_inet_addr_of_string
called undefined primitive caml_gr_sigio_signal
called undefined primitive caml_install_signal_handler
http_get uuencode/fixed14.psf
Font 'fixed14.psf' loaded (256 chars) (mapped)
```

Figure 2. O'Browser's early look

See (9) for the official specification. The DOM is an API to manipulate the run-time representation of a Web page.

Each DOM node is an object, in the JavaScript sense, which can be seen as a map associating string keys to *value*s, where values can be references to other nodes, primitive types like strings and integers, or closures. A $(\mathrm{key}, \mathrm{value})$ pair of an object is usually called a *property*, or a *method* if the value is a closure. To be precise, JavaScript method calls use the following mechanism: the free variable ⌜**this**⌝ in the body of a function f definition is instantiated to o when f is bound to m in the object o and called with the syntax $o.m(args)$. The JavaScript syntax ⌜f. call (o, args)⌝[4] is also available and equivalent.

The whole DOM can then be seen as a set of references to DOM nodes, with a few primitives to access, modify and remove elements of the nodes:

$$\begin{aligned} \mathrm{get} &: \mathrm{node\ ref} \rightarrow \mathrm{string} \rightarrow \mathrm{value} \\ \mathrm{set} &: \mathrm{node\ ref} \rightarrow \mathrm{string} \rightarrow \mathrm{value} \rightarrow \mathrm{unit} \\ \mathrm{unset} &: \mathrm{node\ ref} \rightarrow \mathrm{string} \rightarrow \mathrm{unit} \end{aligned}$$

In this model, each page element (each markup in the XML file) is represented by a node as follows: the *tagName* key indexes the name of the XML element, and each attribute is bound to its value. The children of a markup n are of course also represented as nodes, and a reference to each of them appears in n. For example the DOM node n representing the HTML code ⌜<**img src**="img.png" />⌝ verifies $n(\text{“tagName”}) = \text{“IMG”}$ and $n(\text{“src”}) = \text{“img.png”}$. In fact, the DOM does not contain only XML nodes, but can also contain user-created values.

An important property is that an XML node of the DOM can be referenced by at most one other XML node. In practice, if one takes a node n referenced by the node n_a and makes it a child of n_b with the set operation defined above, then n immediately disappears from n_a.

With such a property, the DOM is able to describe the tree structure of an XML document, allowing its modification while avoiding sharing and cycles of page elements (sharing and cycles are however not forbidden for the data which do not represent the Web page, like the environment of closures, user-created nodes, etc.).

In practice, the browser exposes the DOM of the current page to the JavaScript API through a few global variables. For example, the ⌜document⌝ variable represents the root of the HTML content.

DOM assignments As we have just said, assigning a node reference to a key in another node makes it disappear from other nodes. In a similar way, assignments sometimes have side effects. For example, modifying the ⌜" style "⌝ property of a node forces the browser to recompute the appearance of the associated page element. Similarly, assigning the ⌜"innerHTML"⌝ property makes the browser parse the given string as HTML and replace the children of the node.

DOM events As we have said, nodes need not only contain simple types and node references, but also closures. When some event named e occurs to a DOM node n in the browser, and if the key e is bound to a closure in n, then $n(e)$ is called. More practically, listing 2 shows how to greet the user when the page is clicked. In JavaScript, closures are appended to an *event queue*, so have to wait for the termination of all pending events handlers to be executed.

```
window.onclick = function () {
  document.innerHTML = "<h1>Hello!</h1>";
}
```

Listing 2. Assigning a handler to an event in JavaScript

Accessing the DOM from O'Browser We provide an Objective Caml ⌜Node⌝ module, containing the DOM primitives. A simplified signature is given in listing 3.

```
module Node : sig
  type t
  (* the root node *)
  val document : t
  (* create nodes and text *)
  val text : string -> t
  val element : string -> t
  (* attributes (string properties) *)
  val get_attribute : t -> string -> string
```

[4] In fact, every function is an instance of the built-in object type ⌜Function⌝ which defines the methods ⌜call⌝ and ⌜apply⌝.

```
  val set_attribute : t -> string -> string -> unit
  val remove_attribute : t -> string -> unit
  (* events *)
  val register_event
    : t -> string -> (unit -> unit) -> unit
  val clear_event : t -> string -> unit
  (* access node's children *)
  val children : t -> t list
  val append : t -> t -> unit
  val remove : t -> t -> unit
end
```

Listing 3. Node: The low level DOM interface

The type ⌜Node.t⌝ defines an abstract value containing a JavaScript DOM node reference. Functions are provided to do the primitive operations described above:

- get the root node,
- create fresh nodes (XML element nodes as well as text nodes),
- set properties of the three types:
  - get, set and remove attributes (string properties),
  - link (and unlink) caml closures to events,
  - access and modify child nodes.

Listing 4 is a toy example use of the ⌜Node⌝ module. It is a recursive function collecting the URLs of all the links in the page. As said earlier, the ⌜"tagName"⌝ property of a node is its XML element name, for links, it is ⌜"A"⌝. The attribute ⌜"href"⌝ of a link markup contains its target (like in ⌜<**a** **href**="index.html">index</**a**>⌝).

```
let get_urls () =
  let r = ref [] in
  let rec get_urls node =
    match get_attribute node "tagName" with
      | "A" -> r := get_attribute node "href" :: !r
      | _ -> List.iter get_urls (children node)
  in
    get_urls document ; !r
```

Listing 4. Node usage example

Higher level HTML content creation Using only the DOM primitives to create page elements can be compared to using assembly language to produce complex software. Hence, we provide a higher level interface, with the ⌜Html⌝ module. This module defines a function for each HTML markup, using Objective Caml's labelled optional arguments[5] for optional attributes. When using the ⌜Node⌝ module, to construct nodes with children, one has to create an empty node and then add its children one by one. To simplify this, the ⌜Html⌝ functions for elements with children take a list of nodes as argument. Similarly, each function takes an optional list of $(\mathrm{string} \times \mathrm{string})$ values to define additional attributes.

```
(* signature *)
val string :
    string -> Node.t
val a :
  ?href:string -> ?name:string ->
  ?style:string -> ?onclick:(unit -> 'a) ->
  ?attrs:(string * string) list ->
  Node.t list -> Node.t
(* example <a href="index.html">index</a> *)
let link = a ~href:"index.html" [string "index"]
```

Listing 5. Link creation with Html

Listing 5 shows an extract of the signature of the module ⌜Html⌝, along with an example use.

[5] The syntax ⌜?a : t⌝ defines an optional argument a of type t, taking the value ⌜Some x⌝ if x is passed to the function via ⌜~a:x⌝ or ⌜None⌝ if a is omitted.

Compatibility The virtual machine itself has been successfully ported to the JavaScript engines of all the current common browsers, namely Microsoft Internet Explorer 6+, Mozilla Firefox 2 and 3, Opera 9+, Apple Safari 3.0+ and Konqueror 3.5+. This required to compare the efficiency of JavaScript's control flow structures. For instance, we did a little benchmark between JavaScript's ⌜**switch**⌝, an array of functions and nested ⌜**if**⌝s. The result is that we use an array of functions indexed by instruction codes instead of a switch since this approach has performs better on some engines and similarly on others. We made the arguable choice to keep a single code for all platforms, but a solution would be to provide several implementations of the same algorithm using different coding styles and to choose the best one at run-time.

However, even if the core is compatible, and since we provide a low level DOM interface and there are differences between the DOMs exposed by the different browsers, the same application may run but have different behaviours. A first solution we propose (and are still working on) is to solve these issues in the higher level ⌜Html⌝ module. For instance, Microsoft Internet Explorer 6 does not support the ⌜"position : fixed"⌝ style attribute used to specify that a div stays visible even if the page is scrolled. We thus provide a ⌜~fixed⌝ optional argument to the ⌜Html.div⌝ function which simulates this behaviour with JavaScript but uses the style attribute on other browsers. Another solution would be to abstract the styles by Objective Caml types.

3.4 A compatible data representation

Even if Objective Caml allows the definition of quite complex data types, the run-time representation of values is limited to two cases: integers and pointed blocks, with a small number of block types. In standard Objective Caml, on a n bit architecture, a value can be either a $n-1$ bit integer, encoded in the higher part of a n bit word, the least significant bit being a 1, or a pointer to an even address, the last bit being by definition a 0. The discrimination between integers and pointers is then simply a done by testing the least significant bit. If the value is a pointer, then the pointed data can be mainly an array of values, a closure, a boxed 64 bit float, an array of unboxed 64 bit floats or a foreign value encapsulated in a *custom* block[6].

We chose to remain as close as possible to the standard representation, in particular, we can discriminate if a value is an integer or a block, use the same boxing policy for floats and float arrays and implemented the same standard *custom* blocks for the standard library, like ⌜Int64⌝.

Having a closely similar data representations is really important for the following reasons:

- We can (de)serialize values in a compatible format. This is important since we want to be able to transmit data between the browser and the server as Objective Caml values through serialization.
- Some Objective Caml features, like a large part of the Object Oriented layer, are implemented in the language itself, sometimes by bypassing the type inference, and rely on the data representation. By using a compatible representation, we can use the standard library unmodified and so save time and avoid the bugs of a new implementation.
- It also helped us debug the virtual machine by comparing the data structures.

Implementation In practice, an integer is simply a JavaScript ⌜Number⌝ [7], and a pointer is an instance of the object type ⌜Block⌝.

[6] A block containing the foreign value along with a pointer to its associated primitives for hashing, comparison, serialization, etc.

[7] Simple types are not a special case: one can for instance write ⌜(3). toString ()⌝ in JavaScript.

Then the discrimination between the two is done by the test ⌜(b **instanceof** Block)⌝. The definition of the object type ⌜Block⌝ is presented in listing 6. A block encapsulates an array of values, which can be either integers or blocks, thanks to the permissive type system of JavaScript. To be able to mimic some of the behaviours of the original run-time on values, the prototype of the ⌜Block⌝ type provides a ⌜shift⌝ method which simulates pointer arithmetic[8]. This method creates a new block b from a block a, sharing its content with a but for which the ⌜get⌝ and ⌜set⌝ operations use an offset.

```
function Block(size, tag) {
  // functions can be seen as object constructors
  // (new Block(s,t)) creates an object of type Block
  // "this" is a reference to the new object
  this.size = size;
  this.tag = tag;
  this.content = [];
  this.offset = 0;
}
// modifying the prototype of the function Block makes
// its content accessible from all objects of type Block
// (this is how inheritance is done in \js)
Block.prototype.get = function (i) {
  return this.content[this.offset + i];
}
Block.prototype.set = function (i, v) {
  this.content[this.offset + i] = v;
}
Block.prototype.shift = function (o) {
  var nsize = this.size - o >= 0 ?this.size - o :0;
  var b = new Block (nsize, this.tag);
  b.content = this.content;
  b.offset = this.offset + o;
  return b;
}
```

Listing 6. The Block object

Foreign functions interface Since we use Objective Caml's byte-code calling convention and a data representation different from JavaScript for simple types and objects, it is not possible to call existing JavaScript functions from Objective Caml directly. To be able to use existing JavaScript libraries, one needs to write interface code (as it is the case with the C language for standard Objective Caml programs).

To achieve this, the virtual machine exposes a ⌜RT⌝ global object which is used for the look-up of foreign functions' symbols. To define a foreign function, the programmer associates a function to its symbol in ⌜RT⌝. Thanks the JavaScript methods ⌜call⌝ and ⌜apply⌝ of the ⌜Function⌝ prototype, a foreign function is simply defined as a JavaScript function taking the same number of parameters as the Objective Caml associated function. To call such a function, the virtual machine then simply does a ⌜RT["symbol"]. call (vm, args)⌝. The first argument of ⌜call⌝ is the current virtual machine state, which, for instance, enables the programmer to raise exceptions in the virtual machine. This is important since, as said earlier, multiple instances of the virtual machine can run simultaneously.

Listings 7 and 8 shows how the ⌜String . get⌝ function is declared and defined.

```
external get
  : string -> int -> char
  = "caml_string_get"
```

Listing 7. A foreign function declaration

```
RT.caml_string_get = function (arr, idx) {
    if (idx >= 0 && idx < arr.size - 1) {
        return arr.get(idx);
    }
        this.array_bound_error ();
}
```

Listing 8. A foreign function definition

[8] This is used for instance to support Objective Caml's representation of mutually recursive functions, and to handle code segments as blocks.

3.5 The execution model

To fulfill our goal to have a unique language to program a complete Web application, we wanted the execution model to be as close as possible to the one of a standard Objective Caml environment.

In particular, the JavaScript execution model is such that the browser is blocked during a JavaScript computation. The JavaScript programmer has to give back the control to the browser explicitly to keep the user interface reactive. Moreover, there is no easy yield operation: a yield is done by creating manually a continuation and by adding its execution to the event queue through a call to ⌜window.setTimeout(0, continuation)⌝. When compiling to JavaScript, a solution is to rewrite the program in CPS form, then a yield is done by putting the current continuation in the event queue.

We did not want to add an explicit yield operation to the language and thus change the programming style. So the solution is the following simple behaviour: the virtual machine executes a quota of instructions and then puts in the event queue a closure which runs itself again. Since foreign functions may take an undetermined time, a time barrier is also set. To sum up, the programmer just writes normal Objective Caml code, and the virtual machine handles all the low level control flow passing operations.

Blocking calls Some functions, like ⌜wait_next_event⌝ from the standard library graphical module, or the ⌜http_get⌝ function provided by O'Browser which gets the content of a file from its URL, are seen as blocking calls from the Objective Caml point of view. However, it would be a poor solution to block the browser while waiting for the resource. The solution we adopted is to freeze the machine as follows:

1. When a foreign function[9] wants to wait for a resource, it stores the continuation of its execution into a closure c_{res}.
2. It then tells the virtual machine, by raising a JavaScript exception, that it has not finished and is waiting for the resource r.
3. The machine assigns to the event spawned when r is ready a closure which resumes itself, and then stops.
4. When the virtual machine is run again, it first checks if some c_{res} is present and runs it.
5.
   - If c_{res} succeeds, then the machine puts the result on the stack as it would be for a normal foreign function call, and continues to execute the remaining byte-code.
   - If c_{res} does not succeed, or if it simply has to wait for another resource before returning, the same process is done one more time.

Actually, this is a simplification, and the real implementation involves more states and resource identifiers, but these are technical details.

```
for i = 0 to 10 do
  alert (sprintf "iteration␣%d" i) ;
  sleep 1
done
```

Listing 9. Blocking calls example

An interesting gain compared to JavaScript is the possibility to resume a computation after a given time, without the need to

[9] The JavaScript implementation of an Objective Caml external function.

construct manually the continuation, as shown in listing 9. With our implementation, the virtual machine is stopped during the ⌜sleep⌝ call and the browser thus stays reactive.

Concurrency O'Browser implements the ⌜Threads⌝ module of the Objective Caml standard library. This is simply done by putting the virtual machine register in a context object, and by wrapping the byte-code interpretation loop within a scheduler which regularly does a context switch.

Some other tricks were needed, for example the blocking calls presented above do not actually stop the machine but try to run another thread and only stops if no thread is alive.

Not surprisingly, if the implementation was conceptually simple to implement, it was hard to debug due to the number of states and the difficulty to handle the control flow in JavaScript). Finding a good scheduling algorithm and its associated constants took a lot of time. The problem is that it has to run enough instructions for the program not to run too slowly while doing enough context switch to provide concurrency. It also has to pass the control flow to the browser often enough so that the user interface stays reactive, but not too often because this operation has a high and quite unpredictable impact on performance. The current one seems good enough on most of our examples but we are still tweaking it.

Events handlers as concurrent threads We chose to use a different semantics than JavaScript's one for events execution. While in JavaScript, event handlers are executed sequentially by putting them in a queue, in O'Browser, each event handler is launched in a separate thread. The positive result is that the programmer may write an arbitrary long computation in an event handler, while keeping the rest of the application reactive. However, one cannot assume that, for instance, a ⌜mouseOver⌝ event has terminated when its corresponding ⌜mouseOut⌝ event is launched. Of course, these issues are solvable by using Objective Caml's Mutexes, implemented by O'Browser.

4. Experimental results

We now present the results of our experiment, first by comparing its performance to Objective Caml and JavaScript, and then by showing some demonstrative examples to put the, not surprisingly, quite poor benchmark results into perspective.

4.1 Performance

Comparison to other technologies We did a comparison between ocamlc (the standard byte-code compiler), ocamljs (an experimental compiler to JavaScript (14)), smltojs (a compiler of Standard ML to JavaScript (16)), JS (an equivalent of the algorithm in plain JavaScript if we were able to write one) and O'Browser.

- *ack* is the Ackermann function called on $(3, 6)$ twenty times.
- *kb* is the Knuth-Bendix rewriting algorithm.
- *queens* is a solver for the n-queens problem.
- *nucleic* is a benchmark using float and trees used in (P. H. Hartel, M. Feeley *et al* 1996) to compare functional language implementations

The numbers are execution times, on a mainstream Intel Dual Core platform running a 64 bit GNU/Linux operating system. The browser used for the tests is Mozilla Firefox 3, except for *kb* since the smltojs version made Firefox completely hang for some unknown reason, so we used Opera 9.5.

test	ocamlc	JS	ocamljs	smltojs	O'Browser
ack	$160ms$	$1.6s$	$5s$	$3s$	$1m20s$
kb	$1s$	-	fails[10]	$> 1h30$	$9m15s$
queens	$75ms$	fails[11]	$2s$	$3s$	$21s$
nucleic	$800ms$	-	$2m8s$	$32s$	$5m15s$

If we look at the raw data, the performance is poor. However, we chose these examples because they show the three common cases and allow us to put these bad results into perspective:

- *ack* is a computation intensive task, with a unique recursive function over integers, so is a function quite easily optimizable by the JavaScript interpreters (integers unboxing, etc.), whereas the abstraction of the virtual machine prevents such optimizations. By the way, it is important to notice that the JavaScript tests (JS and ocamljs) didn't run at our first try and required us to tweak the interpreter since the number of recursions is limited to usually 100 or 1000 (since the virtual machine has its own stack, O'Browser does not suffer from this limitation).
- *kb* is complex, highly functional, using exceptions as a programming style and doing many string manipulations. The compilation of such a program is hard and indeed breaks ocamljs and is extremely slow in smltojs (that is not due to the syntax conversion since the Standard ML version, compiled by mlton, is even faster than the Objective Caml one compiled by ocamlopt). So, even if we get a big ratio between the two Objective Caml virtual machines, this is a case where the abstraction provided by O'Browser is a clear gain.
- *queens* and *nucleic* show the common case of the performance ratio, which is around 10 to 20 between JavaScript (or compiled to JavaScript) code and O'Browser.

An important result is that even if the ratio fluctuates between the two virtual machines, it seems more likely to stay in a predictable range since the algorithms are similar; such a result could hardly be obtained with a compiler using JavaScript control flow primitives (like the exception mechanism), as demonstrated by the *kb* example.

Moreover, the examples of the next section will show that this factor stays acceptable for many use-cases in client side Web programming. In fact the time is spent more in user interaction and page redrawing than in code execution (as it is the case in the computational examples from this benchmark) .

Comparison between JavaScript engines Not surprisingly, since JavaScript engines' performance is a central point in the fight between browsers, the execution times in most of the popular browsers are similar. During our tests, the order (beginning with the fastest) was Firefox 3, Safari 3.2, Opera 9.5, MSIE 6, Firefox 2, and the speed factor between the fastest and the slowest was about 2. There is then a gap between the aforementioned ones and MSIE 8 and Konqueror, but the former one is still in beta stage, and the Konqueror developers have just announced the integration of a new engine so the situation will probably completely change in a near future.

As previously said, computation time is not often the major problem in current Web applications, and it is a good result for portability to see that none of the common browsers is left behind. Another positive result is that Opera has good performance, since it is almost always the browser embedded in gaming platforms and set top boxes.

[10] The compiler succeeds but the generated code behaves incorrectly.

[11] We wrote a version in JavaScript but the lack of tail recursion prevented it from running with our available memory.

4.2 Examples

We now present some concrete uses of O'Browser. We start with a small overview of the two main aspects of client side scripting, namely the creation of new HTML content and the modification of the page, along with a presentation of how to do this in O'Browser. Then we present two small yet quite complex and demonstrative Web applications written entirely in Objective Caml. These examples are extracted form the O'Browser tutorial, and are thus available for testing by the reader.

Content creation One possible use of O'Browser is to collect the raw data through HTTP requests and to create an HTML view of it. This can can prove useful to define page data without having to encode it in HTML at each modification, and without requiring a server script.

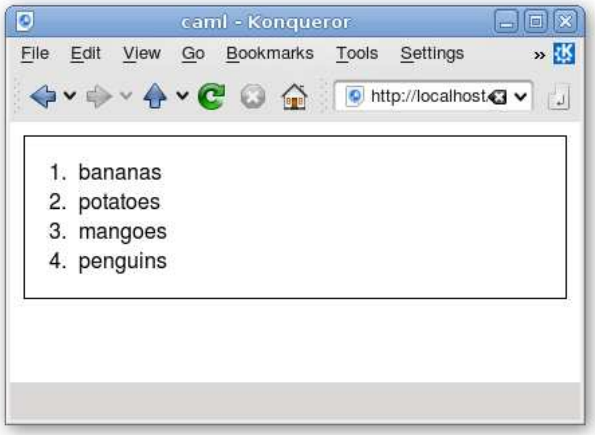
For example, listings 10, 11 and figure 3 show the different parts of a small code snippet creating an HTML ordered list from the lines of a text file, and its rendering in a browser.

```
let items = split (http_get "list.txt") '\n' in
let container = get_element_by_id "basket" in
  Node.append
    container
    (Html.ol
       (List.map
          (fun n -> Html.li [Html.string n])
          (List.filter ((<>) "") items)))
```

Listing 10. A list from a file: O'Browser code

```
<body>
  <div id="basket" style="border:1px␣black␣solid"></div>
</body>
```

Listing 11. A list from a file: html code (extract)



1. bananas
2. potatoes
3. mangoes
4. penguins

Figure 3. A list from a file: an execution

Page modification Another common use of client side scripting, is the modification of the page to use a more dynamic interface when JavaScript is enabled, while keeping the same content.

There are several possibilities to find out which DOM elements are to be modified. One can for instance insert recognisable patterns in CSS classes, in comments or even simply add custom (not XHTML valid) markups or attributes. Our solution is to encode *commands* in markups ids. This has the advantage of keeping the XHTML code valid since every markup can have an id attribute. We use an encoding of forbidden characters using colons (which is one of the few special characters authorized in ids), and we provide the encoder and decoder. A command is an Objective Caml list of strings, and is encoded as the concatenation of these strings separated by two colons with forbidden characters encoded as entities surrounded by colons. For example, the command ⌜["view";"imgs/13.jpg"]⌝ is encoded as ⌜"uid0::view::imgs:slash:13.jpg"⌝. A unique string is prefixed, since all ids in an XML document must be different.

For instance, in the tutorial, we browse the DOM to find code fragments and apply a syntax coloration. Listings 12 and 13 show respectively the O'Browser code to detect source code extracts to colorize, and a colorization command encoded in an id.

```
match
  get_attribute node "tagName",
  decode_id (get_attribute node "id")
with
  | "span", ["inline-source" ; lang] ->
    let code = get_attribute node "textContent" in
      colorize node code lang
  | "div", ["source" ; lang ; file] ->
    let code = http_get file in
      colorize node code lang
  | (* interpret other commands ... *)
```

Listing 12. Inline code coloration: O'Browser code

```
<span id="uid0::inline-source::ocaml">let x = 2</span>
<div id="uid1::source::ocaml::test:dot:ml"></div>
```

Listing 13. Inline code coloration: html code (extract)

We are still working on DOM interaction and the code may neither seem very clean nor beautiful. However, it foreshadows the efficiency of Objective Caml over JavaScript, in particular, it shows how the pattern matching and functional style help the programmer introspect and interact with the document in a few lines of code.

Client/server over http example Using plain Objective Caml (with the sockets from the standard library) to program a simplified HTTP server, and O'Browser (with its ⌜Graphics⌝ implementation), we wrote a simple *proof-of-concept* communicating Web application entirely in Objective Caml.

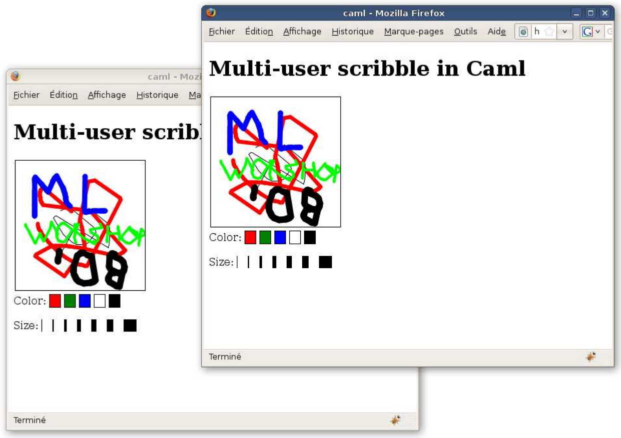

caml - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
Multi-user scribble in Caml
Color:
Size:
Terminé

Figure 4. A multiuser scribble in Objective Caml

Figure 4 shows a session of this toy application which enables several users to share a whiteboard over the network through their Web browsers. The simplicity of the Web server and the number of requests due to the naive textual-over-http protocol makes it a little slow, but it works. The whole application fits in a little more than a hundred lines of Objective Caml.

Integration with Ocsigen Even if O'Browser cannot (yet) directly communicate with Ocsigen, it is already possible to make it communicate via standard http request, and to use it to enhance Ocsigen modules.

For instance, we wrote a little image gallery script. The Ocsigen part takes a directory of images, generates thumbnails and re-sized images. The main service generates a page of thumbnails linked to the pictures, with encoded commands to call the viewer on the re-sized pictures (as presented earlier in the section).

When the gallery is loaded, the O'Browser programs searches for the encoded commands in the DOM and replaces the links' targets to calls to a caml closure launching a picture browser in the same window.

The picture browser then dynamically loads and displays the re-sized pictures, providing buttons to previous and next pictures (since the list of all pictures is collected during the initial DOM search), a screen capture is shown in picture 5.

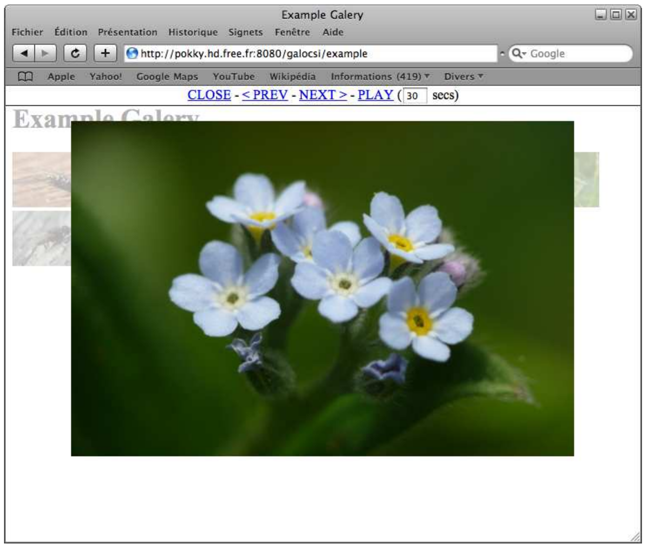

Example Galery
Fichier Édition Présentation Historique Signets Fenêtre Aide
http://pokky.hd.free.fr:8080/galocsi/example
Google
Apple Yahoo! Google Maps YouTube Wikipédia Informations (419) Divers
CLOSE - < PREV - NEXT > - PLAY (30 secs)
Example Galery

Figure 5. A gallery in Ocsigen and O'Browser

The advantage of such an approach is that the original page remains accessible, since it is valid XHTML as checked by the type system of Ocsigen and fully working even when JavaScript is not available.

Again, the whole application fits in a little more than a hundred lines of Objective Caml.

5. Related work

As there are lots of Web programming tools, we focus here on the ones that generate code to be executed on the client, and especially those inspired by the functional programming community.

Client/server frameworks A few projects share Ocsigen's goal of generating code for both server and client side (and also for database queries) using one language.

One of the most closely related is HOP (Serrano 2007) (17), a language inspired by Scheme, with two compilers for producing server and client side code. It is using a two level language to distinguish between server and client portions of code, the latter being compiled towards JavaScript. The Web application is therefore written in one single code, and the two execution flows communicate through function calls.

Links (18) is also a language allowing to write server and client sides in the same code, through annotations. It is using a compiler towards JavaScript, with a concurrency model that uses message passing in Erlang or Mozart style (19) (20). Communication between server and client is done through function calls.

Client side code execution Another compiler towards JavaScript is Ycr2js (21) (Yhc Core to Javascript Converter) that converts Haskell to Javascript. It implements concurrency through a cooperative thread model.

There are also other compilers to JavaScript, for example in the GWT(13) or Volta (22) frameworks.

Flapjax (23) is a language to program the client side of a Web application. It is using reactive programming. The language uses JavaScript syntax, and a compiler to translate the reactive part into JavaScript. It is also possible to use it simply as a library in JavaScript.

We are not aware of any other virtual machine implemented in JavaScript, but one virtual machine for Java called Orto (24), for which we have very little information. Some interpreters for scripting languages have also been written in JavaScript.

6. Conclusion

Thanks to this experiment, our goal to write a Web application entirely in Objective Caml is one step closer. We were able to write the server code, thanks to Ocsigen, now, we have a fully functional, yet a bit slow, experimentation platform to write the client code.

Future work We plan to work on the communication between the two parts. As explained in section 3, we use a compatible run-time data representation which enables us to transmit serialized data. We plan to use the work from (Henry et al. 2007) on safe deserialization of Objective Caml values to design our security model.

Moreover, we have started to work on statically type-checked database interfacing for the server side. The goal is to ensure that the program uses a coherent database scheme before running it and thus eliminate run-time errors. Some work has already been done by pgocaml (25), we plan to add more flexibility to the model, to add a modular back-end interface to support several database engines, and to perform the checks at the deployment phase instead of the compilation.

We also plan to ensure properties on the preservation of the XML grammar of the DOM during its modification. Several works have been done on XML typing integration in Objective Caml, like OcamlDuce (Frisch 2006) (26), but they are centered on XML tree generation. It does not seem trivial to apply them directly to the DOM due to its dynamic nature, in particular the fact that a side effect on a node of the DOM may affect other nodes and thus break their well typedness.

Nearer future work There are two visions of client side Web programming. The first, which is the most widespread one today, is to enhance Web sites by adding a more reactive user interface. For this, and as shown by the examples presented in section 4, O'Browser has good enough performance and using Objective Caml gives a better programming experience in terms of safety and productivity.

The second one is RIA (*Rich Internet Applications*) which are arising with tools such as Google SpreadSheets and show the limits of JavaScript (the reader may have a look at such applications to be convinced that there are performances and debugging issues with them). For this kind of applications, we have already started to write a browser plug-in embedding the Objective Caml virtual machine and exposing the same Objective Caml API as O'Browser. On one hand, this will give a great performance gain to users who install the plug-in, on the other hand, users who do not want, or cannot, install the plug-in will still be able to run the applications thanks to our JavaScript implementation.

Links

(1) O'Browser tutorial
http://www.pps.jussieu.fr/~canou/obrowser/tutorial/

(2) Ocsigen and Eliom
http://www.ocsigen.org/

(3) MMM, a web browser in Objective Caml
http://pauillac.inria.fr/~rouaix/mmm/papers/

(4) The embedded Objective Caml top-level
http://www.pps.jussieu.fr/~capel/eng/toplevel/toplevel.html

(5) MLj, an ML to Java compiler
http://www.dcs.ed.ac.uk/home/mlj/

(6) The Ocaml-Java project
http://ocamljava.x9c.fr/

(7) SML.NET, an SML to .Net compiler
http://www.cl.cam.ac.uk/research/tsg/SMLNET/

(8) OCamil, an Objective Caml to .Net compiler
http://ocamil.ortsa.com

(9) The DOM (Document Object Model)
http://www.w3.org/DOM/

(10) The Java DOM API
http://www.mozilla.org/projects/blackwood/dom/

(11) The Java plug-in
http://java.sun.com/products/plugin/1.3/docs/jsobject.html

(12) Comparison of JavaScript libraries
http://en.wikipedia.org/wiki/Comparison of JavaScript frameworks

(13) Google Web Toolkit
http://code.google.com/webtoolkit/

(14) Ocamljs, an Objective Caml to JavaScript compiler
http://skydeck.com/blog/programming/ocamljs-ocaml-to-javascript-compiler/

(15) Afax, F# web tools
http://tomasp.net/projects/fswebtools.aspx

(16) SMLtojs, an SML to JavaScript compiler
http://www.itu.dk/people/mael/smltojs/

(17) Hop, a web 2.0 programming language
http://hop.inria.fr/

(18) Links, a web 2.0 programming language
http://groups.inf.ed.ac.uk/links/

(19) Erlang, a language for distributed and concurrent applications
http://www.erlang.org/

(20) Mozart, a language for distributed and concurrent applications
http://www.mozart-oz.org/

(21) Ycr2js, an Haskell to JavaScript compiler
http://www.haskell.org/haskellwiki/Yhc/Javascript

(22) Volta, web applications in .Net
http://labs.live.com/volta/

(23) Flapjax, a reactive client side web progamming language
http://www.flapjax-lang.org/

(24) Orto, a JVM in JavaScript
http://orto.accelart.jp/

(25) pgOcaml, typed database requests for Objective Caml and PgSQL
http://developer.berlios.de/projects/pgocaml/

(26) OcamlDuce, XML typing for Objective Caml
http://www.cduce.org/ocaml

References

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

Vincent Balat. Ocsigen: Typing web interaction with objective caml. In *ML'06: Proceedings of the 2006 workshop on ML*, pages 84–94, Portland, Oregon, USA, 2006. ACM. ISBN 1-59593-483-9.

Nick Benton and Andrew Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *Proceedings of the International Conference on Functional Programming (ICFP)"*, 1999.

Nick Benton, Andrew Kennedy, and George Russel. Compiling Standard ML to Java Bytecodes. In *Proceedings of the International Conference on Functional Programming (ICFP)*, September 1998.

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 51–63, 2003. ISBN 1-58113-756-7.

Emmanuel Chailloux, Grégoire Henry, and Raphael Montelatici. Mixing the objective caml and c# programming models in the .net framework. In *Multiparadigm Programming with OO Languages (MPOOL'04)*, June 2004.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *5th International Symposium on Formal Methods for Components and Objects*, November 2006.

David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998. ISBN 1565923928.

Alain Frisch. Ocaml + xduce. In *Proceedings of the international conference on Functional programming (ICFP)*, pages 192–200. ACM, 2006.

Grégoire Henry, Michel Mauny, and Emmanuel Chailloux. Typer la désérialisation sans sérialiser les types. *Technique et Science Informatiques*, 26(9):1067–1090, November 2007.

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

Xavier Leroy and Jean-Vincent Loddo. Functional programming for the web, Apr 2003. Hyperlearning – Project proposal for the 6th framework program of the European Union, http://www-lipn.univ-paris13.fr/ loddo/funding/projet-hyper-learning.pdf.

Xavier Leroy and François Rouaix. Security properties of typed applets. In J. Vitek and C. Jensen, editors, *Secure Internet Programming – Security issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 147–182. Springer, 1999.

Xavier Leroy, Didier Rémy with Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The objective caml system release 3.10 documentation and user's manual. Technical report, Inria, may 2007.

Raphael Montelatici, Emmanuel Chailloux, and Bruno Pagano. Objective caml on .net: The ocamil compiler and toplevel. In *3rd International Conference on .NET Technologies*, May 2005.

Andrew Kennedy Nick Benton and Claudio Russo. Adventures in Interoperability: The SML.NET Experience. In *Proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, August 2004.

P. H. Hartel, M. Feeley *et al*. Benchmarking implementations of functional languages with "Pseudoknot' ', a float-intensive benchmark". *Journal of Functional Programming*, 1996.

François Rouaix. A web navigator with applets in caml. *Comput. Netw. ISDN Syst.*, 28(7-11):1365–1371, 1996. ISSN 0169-7552.

Manuel Serrano. Programming Web Multimedia Applications with Hop. In *Proceedings of the* ACM *Sigmm and* ACM *Siggraph conference on Multimedia*, Best Open Source Software, Augsburg, Germany, September 2007.