# How to Run your Favorite Language in Web Browsers*

## The Revenge of Virtual Machines (over JavaScript)

Benjamin Canou
LIP6 - UMR 7606
Université Pierre et Marie Curie
Sorbonne Universités
4 place Jussieu
75005 Paris, France
Benjamin.Canou@lip6.fr

Emmanuel Chailloux
LIP6 - UMR 7606
Université Pierre et Marie Curie
Sorbonne Universités
4 place Jussieu
75005 Paris, France
Emmanuel.Chailloux@lip6.fr

Jérôme Vouillon
CNRS, PPS UMR 7126
Univ Paris Diderot
Sorbonne Paris Cité
F-75205 Paris, France
Jerome.Vouillon@pps.jussieu.fr

## ABSTRACT

This paper is a concise guide for developers who want to port an existing language to Web browsers, about what to do and what not to. It is based on the various experiments that have been led in the OCaml language community. In particular, it exhibits how reusing the underlying virtual machine and bytecode associated to the language can come of great help in this task.

## Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors

## General Terms

Design, Languages, Experimentation

## 1. INTRODUCTION

In a number of situations, Web developers would prefer not to program the browser in JavaScript[1], but rather in another language, in particular an already existing one. This is the case for instance when porting an existing code base to a live Web demo, in language-based client/server frameworks like GWT[1], HOP[14] or Ocsigen[10], or when a specific application requires the use of an accordingly specific concurrency model or type system (to obtain a higher level of expressivity or safety).

However, the only common denominator between browsers in terms of programming capabilities is JavaScript. Plug-ins used to be the way, but they are not available everywhere anymore. This means, on one hand, that there is no standard way to execute a program written in another language, and, on the other hand, that it has to be done through JavaScript.

We, the authors of this article, come from the OCaml[13] language community, in which several experiments have been led during the past few years to program the client part of Web applications. This experience, along with the fact that OCaml is a language quite different from JavaScript on many points, gives us a good hindsight on this domain, that we want to share. This article is thus intended as a guide for implementors of languages and development environments for the Web. As a more scientific side topic, this article shows how virtual machines come of great help in these solutions, which is a good point since they have been adopted by most recent languages and platforms.

In section 2, we present ways to run core language features. In section 3, we present ways to interact with external JavaScript libraries. In section 4, we explain how to run different concurrency models over JavaScript's event loop.

## 2. EXECUTING THE CORE LANGUAGE

This section describes the possibilities to host the execution of the core structures of your language in JavaScript, and discusses the advantages of virtual-machine approaches. Since we use OCaml to illustrate these techniques, let us start with quick descriptions of JavaScript and OCaml.

*JavaScript and OCaml.* JavaScript is an imperative scripting language, originally aimed as a glue language for more

*Work partially supported by the French national research agency (ANR), PWD project, grant ANR-09-EMER-009-01

[1] In this article, we voluntarily don't speak about ECMAScript, the standard, but JavaScript, the browser specific implementation.


WWW2012 Developer Track, April 18-20, 2012, Lyon, France.

heavyweight components in the browser. It features a very permissive dynamic type system, a just as permissive object model based on prototypes, and some functional capabilities, behind a Java-like syntax. It features very good performance for a scripting language of its class, thanks to impressive recent optimizations of JavaScript engines.

The OCaml language and environment are quite different from JavaScript in many respects. OCaml is a general-purpose programming language, with a very strict and expressive static type system, and brings functional, imperative and object paradigms as first class citizens, along with an expressive module system to organize programs and libraries. The programmer can choose between a bytecode compiler producing portable code for a specific virtual machine, and a native code optimizing compiler.

## 2.1 Compilation to JavaScript

The most obvious way to run a foreign language over JavaScript is of course to write a source to source compiler, from this language to JavaScript. This has been done for OCaml by Jake Donham, in 2007, in the form of OCamlJS[2], a patched compiler extended with a JavaScript backend.

This classical approach is probably the best candidate to achieve good performance while keeping readability and traceability of produced code. Moreover, it enables the implementor to lightly tweak the source language if necessary to fit in the browser's environment.

However, this approach has a few major drawbacks, which in retrospective make the two following approaches preferable when available. A major problem of this approach is probably debugging. Generated code debugging is difficult, JavaScript debugging is also difficult, so the combination of the two can be a real nightmare. It actually took several years and releases for OCamlJS to support the complete language without bugs. Another technical difficulty is semantics preservation. When compiling to JavaScript, one tends to use corresponding core language features, which are sometimes not so corresponding. This is for instance the case in OCaml with exceptions and tail recursion, that cannot be simply translated to JavaScript's equivalent while preserving their performance scheme. From a less technical point of view, for small projects, the mandatory maintenance cost needed to the modified compiler and adapt it when a new version of the original compiler is released can be a major problem.

## 2.2 Bytecode interpretation in JavaScript

An alternative approach was tried by Benjamin Canou in 2008 : OBrowser[11], an OCaml virtual machine (or bytecode interpreter) written in JavaScript. This is an approach we highly recommend to language implementors, at least as a first prototype, since it solves most of the difficulties of a direct compiler as previously presented.

A clear advantage of the approach is its development cost. Indeed, a complete virtual machine and environment can be achieved in a few weeks, and a less than a handful thousands lines of JavaScript code. Moreover, since bytecode formats changes very rarely compared to the associated source languages, maintenance cost is also drastically reduced.

The experiment was focused on compatibility, and managed to maintain a high level of similarity with the original OCaml in terms of semantics. It included for instance a very similar FFI (*foreign functions interface*) and a fully compatible binary serialization format. In terms of compatibility, the approach is a clear win over compilation techniques. Clearly, this gain comes from the virtual machine structure, since the implementor mainly has to ensure the compatibility of a small set of low level instructions, instead of the whole source language. In the same vein, this structure clearly made the debugging of the system easier, by comparing step by step the state of the original OCaml machine with OBrowser's on the same program.

Also, even if the experiment was not focused on performance, OBrowser managed to keep execution times around an order of magnitude (or less) from JavaScript equivalents, a result actually good enough for the vast majority of browser scripting tasks, in which raw computation speed is not the bottleneck, but interaction and rendering. As an example, a Boulder Dash clone written in OCaml and interpreted by OBrowser is available online[3] for the reader to try and see.

## 2.3 Bytecode to JavaScript recompilation

In the continuation of OBrowser, Jérôme Vouillon wrote `js_of_ocaml`, a tool which takes an OCaml bytecode file and, instead of interpreting it directly in the browser, recompiles it to a JavaScript program ahead of time, on the developper's machine.

This method is a compromise between a compiler and a virtual machine. As with the virtual machine approach, starting with the bytecode instead of the source reduces maintenance costs. However, it is quite more complex to write, and as with a compiler from source, it is more difficult to keep the exact same semantics.

The main advantage over a bytecode interpreter is performance. Indeed, by expanding the control flow of the bytecode program to JavaScript code, it can take advantage of recent trace-based optimizations in JavaScript engines. It is also possible to perform optimizations, such as dead code elimination, unboxing or inlining, to obtain better performance. In practice `js_of_ocaml` manages to get execution times close to equivalent native JavaScript programs[4]. To demonstrate the performance, a software rendered animated 3D view of the earth in OCaml recompiled by `js_of_ocaml` is available online[5].

# 3. USING EXISTING JAVASCRIPT CODE

Being able to run any language in a browser is interesting in itself, and may be sufficient in some cases, for instance for live Web demos of existing software. But most of the time, being able to manipulate the Web page is a primary need, as well as being able to use existing, rich and widespread scripting libraries, such as jQuery or Dojo.

The first requirement is to be able to manipulate the Web page dynamically. This is done using the DOM (document object model) : all the elements of the page are reified as predefined JavaScript objects, so that the program can traverse the document tree, delete or add children nodes as well

as modify the content and style by calling predefined methods on these objects. Given this structure, there are two ways to bring DOM manipulation to the hosted language.

The most obvious way is to reuse the existing FFI (*Foreign Function Interface*) provided by most languages, and usually designed to call external C or assembly symbols, to call the aforementioned JavaScript predefined methods instead. To use a more common example than OCaml, a Java port could for instance define an object `Node`, which reuses the JNI mechanism to encapsulate a native JavaScript DOM node and provide a Java method corresponding to each predefined JavaScript method on (`appendChild`, `getElementById`, etc.).

An alternative, more lightweight solution is to provide the hosted language with a small set of primitives sufficient to operate on JavaScript objects (`eval`, `get`, `set`, `call_method`, etc.). However, this method are a possible performance loss, and the fact that it requires a more expressive language than the first solution.

The same two methods can be used to interface JavaScript libraries. However, it can be useful in some cases to define a more high level interface over such low-level bindings, for instance by mapping types or classes to the concepts provided by the library.

# 4. CONCURRENCY MODELS

In many Web frameworks, concurrency is considered a side topic, and handled in an unclear, or at least undocumented way. However, concurrency is a key aspect of numerous programming languages, and should not be overlooked when porting them to the Web browser, since the behaviour of programs may be vastly affected by the use of a different concurrency model. OCaml is such a language, since it supports several, well defined concurrency models. This section describes how we managed to implement these models, and thus are able to maintain the original semantics of the language. Here again, we shall see that starting from bytecode for a virtual machine is a clear advantage.

## 4.1 Concurrency in JavaScript

Before presenting our techniques to implement concurrency models, we have to start with a quick description of the state of concurrency in JavaScript.

*The Event loop.* In JavaScript, concurrency is handled by a classical event loop. Events occurring in the Web page are buffered in a queue, which is emptied at each loop step, by calling sequentially all the associated handlers. After each step, the rendering engine updates the display of the page. As a side effect, a looping script simply blocks the execution, and concurrency has to be handled entirely by hand, by splitting basic code parts in separate functions, and manually transmitting the shared state.

*Web Workers[6].* For many reasons, recent rich client side Web applications tend to rely exclusively on browsers built-in features. However, historically, advanced tasks such as socket-based communications, 3D, etc. where delegated to plug-ins, such as Adobe Flash or Java. In order not to rely on plug-ins anymore, browsers have to catch up on these features. This work is known as HTML5, and primarily consists in the aggregation of proposals made by browser vendors. One of these proposal is known as Web Workers, and is know quite widely available. A Web Worker is simply a thread running JavaScript code, which can be launched from a script inside a Web page, and can only communicate through (string) message passing. It can be useful to write a long computation in direct style, without blocking the user interface. However, it can neither return complex objects (without serializing them) nor interact with the user or the DOM.

## 4.2 Implementing a concurrency model

This section explains how to implement different concurrency models and how to take advantages of recent parallel extensions, in the context of the virtual machine based techniques previously presented.

*Preemptive threads in OBrowser.* Thanks to the simple virtual machine structure, OBrowser is able to simulate preemptive threads. This is an interesting point, since it is the concurrency model provided by most languages. This is performed by transforming the interpreting loop into a scheduling loop, that distributes execution time between threads. Another interesting result is that the scheduling interpreter is able to stop itself to let the browser perform an event loop step, and resume the interpretation afterwards. This makes possible to write an infinite loop in a thread, without blocking the interface completely.

*Cooperative threads with Lwt.* The cooperative threading model is largely adopted in OCaml, in particular through the Lwt[15] library, which encodes concurrency using the functional paradigm. This concurrency model has the advantage to be quite close to the event loop model, and can be compiled to JavaScript easily and reasonably efficiently. It is thus a good model to look at, to obtain a clearer concurrency model than the event loop at a reasonable cost. But it needs an expressive language to be encoded in a concise way and a high level enough virtual machine to be able to recognize basic concurrent operations. For instance, it is possible in `js_of_ocaml` since closure creation is a specific instruction of the virtual machine, allowing to compile them to JavaScript closures that can be fed to the event queue. It is also possible for less flexible languages, but with a little more work, and maybe only from source. Links[12] for instance performs a CPS (*continuation passing style*) code transformation to support its concurrency model on the client.

*Concurrency extensions.* To us, the virtual machine is clearly a good level to implement various concurrency models. But this approach can also come of great help when integrating extensions of the concurrency model, such as Web Workers. First, to be able to integrate such extensions in a way which wont surprise the programmer, having a clear and consistent concurrency model is a key point. As a concrete

example, in the original (Unix) version of Lwt, a primitive is provided to launch a preemptive thread (and wait for its return value) from a cooperative one. This primitive is present to perform blocking calls to external libraries from cooperative programs. Integrating Web Workers is thus simply using this existing behaviour, giving a consistent vision of concurrency to the programmer. Another interesting possibility is the ability to use the virtual machine structure to provide less restricted parallel threads on top of Web Workers. For instance, a byte-code interpreter could use Web Workers to run threads in parallel, and automatically migrate their execution to the main process whenever they have to perform a DOM operation.

# 5. CONCLUSION

We have seen three approaches to use a foreign language in the browser, discussed their pros and cons in terms of core language execution, interoperability and concurrency.

The table below summarizes the advantages of each approach, rating each trait from (+) to (+++).

| | Compiler | VM | Expanser |
|---|---|---|---|
| Simplicity | + | ++ | + |
| Semantics preserv. | ++ | +++ | ++ |
| Maintenance | + | +++ | +++ |
| Performance | +++ | + | ++ |
| Concurrency | ++ | +++ | + |

As a conclusion, we recommend to a programmer willing to port its preferred language to Web browsers not to rewrite a compiler from source, but instead to reuse the bytecode target of its compiler, write a virtual machine in JavaScript, and then write a bytecode to JavaScript expanser if performance problems arise. Another important point is to clarify the concurrency model, and if possible to reuse the original model(s) of the language (and as we've seen, this task is easier at the bytecode level).

This approach has many advantages. First, you get a running prototype reasonably fast thanks to the interpreter method, and you can start writing applications without waiting for the expanser to be complete and optimized. Second, a clear and high level concurrency model gives a better programming experience, and helps integrating HTML5 extensions such as Web Workers or WebCL. And last but not least, the bytecode interpreter method can be used to implement debugging facilities (breakpoints, step by step execution, profiling, etc.), which given the difficulty to debug compiled-to-JavaScript programs is a very interesting feature.

Recently, several other projects have also taken the virtual machine approach to be able to run in the browser binaries produced by existing compilers. We can cite Python[7], Ruby[9] and Java[8] for general-purpose languages, but also more exotic platforms such as old game scripting engines and even game console simulators.

# 6. REFERENCES


[1] http://code.google.com/webtoolkit/.

[2] https://github.com/jaked/ocamljs.

[3] http://www.pps.jussieu.fr/~canou/bdash/.

[4] http://ocsigen.org/js_of_ocaml/performances.

[5] http://ocsigen.org/js_of_ocaml/files/planet/index.html.

[6] http://dev.w3.org/html5/workers/.

[7] http://code.google.com/p/pejs.

[8] https://github.com/nurv/BicaVM.

[9] Ruby on javascript and flash. http://hotruby.yukoba.jp.

[10] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: ocsigen, a web programming framework. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 311–316. ACM, 2009.

[11] B. Canou, V. Balat, and E. Chailloux. O'browser: objective caml on browsers. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canaday, September 21, 2008*, pages 69–78. ACM, 2008.

[12] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, pages 266–296, 2006.

[13] X. Leroy. The Objective Caml system release 3.12 : Documentation and user's manual. Technical report, Inria, 2011. http://caml.inria.fr.

[14] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In P. L. Tarr and W. R. Cook, editors, *OOPSLA Companion*, pages 975–985. ACM, 2006.

[15] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canaday, September 21, 2008*, pages 3–12. ACM, 2008.