Efficient high-level abstractions for web programming
Julien Richard-Foy, Olivier Barais, Jean-Marc Jézéquel

To cite this version:

HAL Id: hal-00920786
https://inria.hal.science/hal-00920786
Submitted on 19 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Efficient High-Level Abstractions for Web Programming

Julien Richard-Foy  Olivier Barais  Jean-Marc Jézéquel
IRISA, Université de Rennes 1
{first}. {last}@irisa.fr

Abstract
Writing large Web applications is known to be difficult. One challenge comes from the fact that the application’s logic is scattered into heterogeneous clients and servers, making it difficult to share code between both sides or to move code from one side to the other. Another challenge is performance: while Web applications rely on ever more code on the client-side, they may run on smart phones with limited hardware capabilities. These two challenges raise the following problem: how to benefit from high-level languages and libraries making code complexity easier to manage and abstracting over the clients and servers differences without trading this ease of engineering for performance? This article presents high-level abstractions defined as deep embedded DSLs in Scala that can generate efficient code leveraging the characteristics of both client and server environments. We compare performance on client-side against other candidate technologies and against hand written low-level JavaScript code. Though code written with our DSL has a high level of abstraction, our benchmark on a real world application reports that it runs as fast as hand tuned low-level JavaScript code.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

Keywords Heterogeneous code generation, Domain-specific languages, Scala, Web

1. Introduction

Web applications are attractive because they require no installation or deployment steps on clients and enable large scale collaborative experiences. However, writing large Web applications is known to be difficult [16, 18]. One challenge comes from the fact that the business logic is scattered into heterogeneous client-side and server-side environments [14, 19]. This gives less flexibility in the engineering process and requires a higher maintenance effort: there is no way to move a piece of code targeting the server-side to target the client-side – the code has to be rewritten. Even worse, logic parts that run on both client-side and server-side need to be duplicated. For instance, HTML fragments may be built from the server-side when a page is requested by a client, but they may also be built from the client-side to perform an incremental update subsequent to a user action. How could developers write HTML fragment definitions once and render them on both client-side and server-side?

The more interactive the application is, the more logic needs to be duplicated between the server-side and the client-side, and the higher is the complexity of the client-side code. Developers can use libraries and frameworks to get high-level abstractions on client-side, making their code easier to reason about and to maintain, but also making their code run less efficiently due to abstraction penalty.

Performance is a primary concern in many Web applications, because they are expected to run on a broad range of devices, from the powerful desktop personal computer to the less powerful smart phone [10, 22]. Using the same programming language on both server-side and client-side could improve the software engineering process by enabling code reuse between both sides. Incidentally, the JavaScript language – which is currently the most supported action language on Web clients – can be used on server-side. Conversely, an increasing number of programming languages or compiler back-ends can generate JavaScript code (e.g. Java/GWT [5], SharpKit1, Dart [8], Kotlin2, ClojureScript [15], Fay3, Haxe [4] or Opa4).

However, using the same programming language is not enough because the client and server programming environments are not the same. For instance, DOM fragments can be defined on client-side using the standard DOM API, but this API does not exist on server-side. How to define a common vocabulary for such concepts? And how to make the executable code leverage the native APIs, when possible, for performance reasons?

Generating efficient code for heterogeneous platforms is hard to achieve in an extensible way: the translation of common abstractions like collections into their native counterpart (JavaScript arrays on client-side and standard library’s collections on server-side) may be hard-coded in the compiler, but that approach would not scale to handle all the abstractions a complete application may use (e.g. HTML fragment definitions, form validation rules, or even some business data type that may be represented differently).

On one hand, for engineering reasons, developers want to write Web applications using a single high-level language, abstracting over the target platforms differences and reducing code complexity. But on the other hand, for performance reasons, they want to keep control on the way their code is compiled to each target platform. We propose to solve this dilemma by providing high-level abstractions in compiled domain-specific embedded languages (DSELs) [7, 11]. Compiled DSELs allow the definition of domain-specific languages (DSLs) as libraries on top of a host language,
and to compile them to a target platform. Their deep embedding gives the opportunity to control the code generation scheme for a given abstraction and target platform.

Kossakowski et al. introduced js-scala, a compiled embedded DSL defined in Scala that generates JavaScript code, making it possible to write the client-side code of Web applications using Scala [13]. However, the authors did not discuss any specific optimization and did not consider performance issues of their approach. Our paper shows how js-scala has been extended to support a set of specific optimizations allowing our high-level abstractions for Web programming to be efficiently compiled on both client and server sides.

We validate our approach with a case study implemented with various candidate technologies and discuss the relative pro and cons of them. We also measured the individual impact of each of our optimizations using micro-benchmarks. Though the code written in our DSL is high-level and can be shared between clients and servers, it has the same runtime performance on client-side as hand-tuned low-level JavaScript code.

The remainder of this paper is organized as follows. The next section overviews the existing approaches defining high-level languages for Web programming. Section 3 presents the framework we used to define our DSLs. Section 4 presents our contribution. Section 5 compares our solution to common approaches. Section 6 discusses our results and section 7 concludes.

2. Related Work

We classified existing approaches providing high-level abstractions for Web programming in four categories, as shown in Figure 1.

![Language engineering processes](image)

**Figure 1.** Language engineering processes

**Fat Languages** The first approach for defining a cross-platform language consists in hard-coding, in the compiler, the code generation scheme of each language feature to each target platform. Figure 1 (a) depicts this process. In order to support a feature related to a specific domain, the whole compiler pipeline (parser, code generator, etc.) may have to be adapted. This approach gives fat languages because a lot of concepts are defined at the language level: general programming concepts such as naming, functions, classes, as well as more domain-specific concepts such as HTML fragment definition. Thus, implementing a fat language may require a high effort and adding support for these languages in development environments may require a even higher effort. Examples of such languages for Web programming are Links [6], Opa, Dart [8].

**Domain-Specific Languages** Another approach consists in defining several independent domain-specific languages [23], each one focusing on concerns specific to a given problem domain, and then to combine all the source artifacts written with these language into one executable program, as shown in Figure 1 (b). Defining such languages requires a minimal effort compared to the previous approach because each language has a limited set of features. On the other hand, it is difficult to have interoperability between DSLs. [24] gave an example of such a domain-specific language for defining Web applications.

**Thin Languages** Alternatively, one can define concepts relative to a specific domain as a library on top of a thin general purpose language (it is also referred to as a domain-specific embedded language [11]). Figure 1 (c) depicts this approach. Defining such a library requires minimal effort (though the syntax of the DSL is limited by the syntax flexibility of the host language) and several DSLs can interoperate freely within the host language. However, this approach gives no opportunity to efficiently translate a concept according to the target platform characteristics because the compiler has no domain-specific knowledge (though some compilers hard-code the translation of some common abstractions such as arrays to leverage the target platform characteristics). Examples of languages following this approach are Java/GWT, Kotlin, HaXe and SharpKit. Libraries written in JavaScript (e.g. jQuery) also match this category though most of them do not support both client and server sides.

**Deeply Embedded Languages** The last approach, shown in Figure 1 (d), can be seen as a middle-ground between the two previous approaches: DSLs are embedded in a host language but use a code generation process. This approach shares the same benefits and limitations as embedded DSLs for defining language units. However, the code generation process is specific to each DSL and gives the opportunity to perform domain-specific optimizations. In other words deeply embedded DSLs bring domain-specific knowledge to the compiler. Js-scala [13] is an example of deeply embedded DSL in Scala for Web programming. It makes it possible to produce JavaScript programs from Scala code that uses basic language concepts like arrays and control structures (if and while) as well as mechanisms specific to the Scala compiler like delimited continuations to handle asynchronous computations. Paper [13] presented the implementation of js-scala using staging, but did not discuss any specific optimization and did not consider performance issues of this approach. In this paper, we show how js-scala has been extended to support a set of specific optimizations allowing our high-level abstractions for Web programming to be efficiently compiled on heterogeneous platforms.

3. Lightweight Modular Staging

This section gives background material on the framework used to define js-scala.

Lightweight Modular Staging [20, 21] (LMS) is a framework for defining deeply embedded DSLs in Scala. It has been used to define high-performance DSLs for parallel computing [3] and to define JavaScript as an embedded DSL in Scala [13].

---

5 The code is available at http://github.com/js-scala
LMS is based on staging [12]: a program using LMS is a regular Scala program that evaluates to an intermediate representation (IR) of a final program. This IR is a graph of expressions that can be traversed by code generators to produce the final program code. Expressions evaluated in the initial program and those evaluated in the final program (namely, staged expressions) are distinguished by their type: a Rep[Int] value in the initial program is a staged expression that generates code evaluating to an Int value in the final program. An Int computation in the initial program is evaluated during the initial program evaluation and becomes a constant in the final program.

Defining a DSL with LMS consists in the following steps:

- writing a Scala module providing the DSL vocabulary as an abstract API,
- implementing the API in terms of IR nodes,
- defining a code generator visiting IR nodes and generating the corresponding code.

4. Efficient High-Level Abstractions for Web Programming

This section presents some tasks typically performed in Web applications, either on client-side or server-side or on both, generalizes them in terms of high-level abstractions, and shows how they are implemented in js-scala to generate efficient code.

4.1 Selectors API

In a Web application, the user interface is defined by a HTML document that can be updated by the JavaScript code. A typical operation consists in searching some “interesting” element in the document, in order to extract its content, replace it or listen to user events triggered on it (such as mouse clicks). The standard API provides several functions to search elements in the HTML document according to their name or attribute values. Figure 2 summarizes the available functions and their differences.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>querySelector(s)</td>
<td>First element matching the CSS selector s</td>
</tr>
<tr>
<td>getElementsById(i)</td>
<td>Element which attribute id equals to i</td>
</tr>
<tr>
<td>querySelectorAll(s)</td>
<td>All elements matching the CSS selector s</td>
</tr>
<tr>
<td>getElementsByName(n)</td>
<td>All elements of type n</td>
</tr>
<tr>
<td>getElementsByClassName(c)</td>
<td>All elements which class attribute contains c</td>
</tr>
</tbody>
</table>

Figure 2. Standard selectors API. The querySelector and querySelectorAll are the most general functions while the others handle special cases.

Listing 1 gives an example of use of various functions from the native selectors API to retrieve a list of input fields within a form. The getWords function first finds in the document the HTML element with id add-user, then collects all itsfieldset children elements, and for each one returns the list of its children elements having class word. The existence of several specialized functions in the API makes it possible to write efficient code, but forces users to think at a low abstraction level.

A high-level abstraction for searching elements in a document could be just one function finding all elements matching a given

function getWords() {
  var form = document.getElementById('add-user');
  var sections = form.getElementsByTagName('fieldset');
  var results = [];
  for (var i = 0; i < sections.length; i++) {
    var words = sections[i].getElementsByTagName('word');
    results[i] = words;
  }
  return results
}

Listing 1. Searching elements using the native selectors API

CSS selector. In fact, most JavaScript developers⁶ use the jQuery library that actually provides only one function to search for elements. Listing 2 shows an equivalent JavaScript program as Listing 1, but using jQuery. The code is both shorter and simpler, thanks to its higher level of abstraction. jQuery provides an API that is simpler to master because it has fewer functions, but this benefit comes at the price of a decrease in runtime performance.

Instead, we propose a solution that has a high-level API but generates JavaScript code using the specialized native API, when possible, in order to get both ease of engineering and performance. We achieve this by analyzing, during the first evaluation step, the selector that is passed as parameter and, when appropriate, by producing JavaScript code using the specialized API, and otherwise producing code using querySelector and querySelectorAll.

def find(selector: Rep[String]) =
  getConstIdCss(selector) match {
    case Some(id) if receiver == document =>
      Document.getElementById(Const(id))
    case _ =>
      SelectorFind(receiver, selector)
  }

Listing 2. Searching elements using jQuery

Our API has two functions: find to find the first element matching a selector and findAll to find all the matching elements. Listing 3 gives the implementation of the find function. It is a Scala function that returns an IR node representing the JavaScript computation that will search the element in the final program. The getConstIdCss function analyzes the selector: if it is a constant String value containing a CSS ID selector, it returns the value of the identifier. So, if the find function is applied to the document and to an ID selector, it returns a Document.getElementById IR node (that is translated to a document.getElementById call by the code generator), otherwise it returns a SelectorFind IR node (that is translated to a querySelector call).

The same applies to the implementation of findAll: the selector passed as parameter is analyzed and the function returns

⁶ According to http://trends.builtwith.com/javascript, jQuery is used by more than 40% of the top million sites.
null references are a known source of problems in programming languages [9, 17]. For example, consider Listing 5 finding a particular widget in the page and then a particular button within the widget. The native querySelector method returns null if no node matched the given selector in the document. If we run this code in a page where the widget is not present, it will throw an error and stop further JavaScript execution. Defensive code can be written to handle null references, but leads to very cumbersome code, as shown in Listing 6.\footnote{However, one could alleviate the syntax burden by using a language such as CoffeeScript [1], that supports a special notation for optional values dereferencing and desugars directly to JavaScript.}

```javascript
var loginWidget =
  document.querySelector("div.login");
var loginButton =
  loginWidget.querySelector("button.submit");
loginButton.addEventListener("click", handler);
```

Listing 5. Unsafe code

```javascript
var loginWidget =
  document.querySelector("div.login");
if (loginWidget !== null) {
  var loginButton =
    loginWidget.querySelector("button.submit");
  if (loginButton !== null) {
    loginButton.
      addEventListener("click", handler);
  }
}
```

Listing 6. Defensive programming to handle null references

Some programming languages encode optional values with a monad (e.g. Maybe in Haskell and Option in Scala). In that case, sequencing over the monad encodes optional value dereferencing. If the language supports a convenient syntax for monad sequencing, it brings a convenient syntax for optional value dereferencing, alleviating developers from the burden of defensive programming.

In our DSL, we encode an optional value of type Rep[A] using a Rep[Option[A]] value, which can either be a Rep[Some[A]] (if there is a value) or a Rep[None.type] (if there is no value). An optional value can be dereferenced using the for notation, as shown in Listing 7, that implements in js-scala a program equivalent to Listing 6. The find function returns a Rep[Option[Element]]. The for expression contains a sequence of statements that are executed in order, as long as the previous statement returned a Rep[Some[Element]] value.

```java
for {
  loginWidget <- document.find("div.login")
  loginButton <- loginWidget.find("submit.button")
} loginButton.on(Click)(handler)
```

Listing 7. Handling null references in js-scala

Such a monadic API brings both safety and expressiveness to developers manipulating optional values but usually involves the creation of an extra container object holding the optional value. In our case, the monadic API is used in the initial program but generates code that does not wrap values in container objects but instead checks if they are null or not when dereferenced. So the extra container object exists only in the initial program and is removed during code generation: Listing 7 produces a code equivalent to Listing 6.
We could use a library on client-side to get a higher level API
the client-side and server-side APIs are very different and that the
using the standard Scala XML library. The reader may notice that
shows how one could implement a similar function on server-side
if the value is null. In the case of the OptionForeach node, it
wraps the code block dereferencing the value within a if checking
that the value is not null.

The IR nodes are not tied to the JavaScript code generator, so
we are able to make this abstraction available on server-side by
writing a code generator similar to the JavaSctipt code generator,
but targeting Scala. So the same abstraction is efficiently translated
on both server and client sides.

4.3 DOM Fragments Definition

This section shows how we define an abstraction shared between
clients and servers, as in the previous section, but that has different
native counterparts on client and server sides. The challenge is to
define an API providing a common vocabulary that generates code
using the target platform native APIs.

```
val articleUi = function (article) {
  var div = document.createElement('div');
  div.setAttribute('class', 'article');
  var span = document.createElement('span');
  var name =
    document.createTextNode(article.name + ': ');
  div.appendChild(name);
  var strong = document.createElement('strong');
  var price = document.createTextNode(article.price);
  strong.appendChild(price);
  div.appendChild(strong);
  return div
};
```

Listing 9. JavaScript DOM creation native API

```
def articleUi(article : Article) =
  <div class="article">
    <span>[article.name + " : "]</span>
    <strong>[article.price]</strong>
  </div>
```

Listing 10. Scala XML API

A common task in Web applications consists in computing
HTML fragments representing a part of the page content. This task
can be performed either from the server-side (to initially respond
to a request) or from the client-side (to update the current page).
As an example, Listing 9 defines a JavaScript function articleUi
that builds a DOM tree containing an article description. Listing 10
shows how one could implement a similar function on server-side
using the standard Scala XML library. The reader may notice that
the client-side and server-side APIs are very different and that the
client-side native API is very low-level and inconvenient to use.
We could use a library on client-side to get a higher level API

for DOM fragment creation, but that would decrease the runtime
performance. Instead, we want to define a high-level API that
compiles to code as efficient as if it was written using the native
APIs on both platforms.

Our first step consists in capturing, in a high-level API, the
concepts common to the JavaScript and Scala APIs. Though they
are different, both APIs define HTML elements with attributes and
content. We propose to have a function el to define an HTML
element, eventually containing attributes and children elements.
Any children of an element that is not an element itself is converted
into a text node. Listing 11 shows how to implement our example
with our DSL. The children elements of an element can also be
obtained dynamically from a collection, as shown in Listing 12.

```
def articleUi(article : Rep[Article]) =
  el('div', 'class' -> 'article')( el('span')(article.name + ': '),
  el('strong')(article.price) )
```

Listing 11. DOM definition DSL

```
def articlesUi(articles : Rep[Seq[Article]]) =
  el('ul')
  for (article <- articles)
    yield el('li')(articleUi(article))
```

Listing 12. Using loops

The el function returns an Element IR node that is a tree com-
posed of other Element and Text nodes. The JavaScript and Scala
code generators traverse this tree and produce code building an
equivalent DOM tree and XML fragment, respectively. When the
children of an element are constant values (as in Listing 11) rather
than dynamically computed (as in Listing 12), the code generators
unroll the loop that adds children to their parent, for better
performance. As a result, Listing 11 generates a code equivalent to
Listing 9 on client-side and equivalent to Listing 10 on server-side.

Listings 13 and 14 show the relevant parts of the code generators
for this DSL. They basically follow the same pattern: they visit Tag
and Text IR nodes and produce the corresponding elements in the
target language.

5. Evaluation

Our goal is to evaluate the level of abstraction provided by our
solution and its performance, by comparing it with common ap-
proaches. We take the number of lines of code as an inverse ap-
proximation of the level of abstraction. We also evaluate the ability
to share code between client and server sides.

We realized two micro-benchmarks involving programs using
the selectors DSL and the optional value DSL, and we bench-
marked a real world program. In each case we have written
several implementations of the program, using plain JavaScript,
Java/GWT, HaXe and js-scala (in each case we tried to write the ap-
lication in an idiomatic way). The performance benchmarks mea-
sured the execution time of the generated JavaScript code. The tests
were executed on a DELL Latitude E6430 laptop with 8 GB of
RAM, on the Google Chrome v27 Web browser.

All our charts show three kinds of measures: the first group is
the speed execution in operations per second (higher is better), the
second group is the number of lines of code (lower is better) and the
last group is the execution speed to number of lines of code ratio
(higher is better). We normalized the values so the three groups can
be shown within a same chart without scale issue.
5.1 Micro-Benchmarks

The micro-benchmarks measure the performance of our implementation of the selectors and optional value abstractions. We could not implement this abstraction in GWT or HaXe as efficiently as we did in js-scala because it relies on the staging mechanism. So we directly compared the execution time of the JavaScript code generated by Listing 4 with a JavaScript program equivalent to Listing 1 but using the high-level native API (querySelector and querySelectorAll). The code was executed in a Web page containing a few elements: 4 fieldset elements, each containing 0 to 2 elements with class word.

Figure 4 shows the benchmark results. The JavaScript-opt version is Listing 1, which uses low-level native APIs, the JavaScript version is the equivalent listing using the high-level native API, and the jQuery version is Listing 2. The js-scala version is slightly slower than the JavaScript-opt (by 14%), but is 2.88 times faster than the JavaScript version, and 28.6 times faster than the jQuery version. Finally, the js-scala version has a performance to lines of code ratio more than 1.72 times higher than others.

5.1.2 Optional Value

We reimplemented the optional value abstraction in plain JavaScript, Java and HaXe and wrote a small program manipulating optional values. Listing 15 shows the js-scala version of this program. The maybe function is a function partially defined on Int values.

Figure 5 shows the benchmark results. The js-scala version of the program runs between 3 to 10 times faster than other versions. The source code of the benchmarks is available at https://github.com/js-scala/js-scala/tree/master/papers/gpce2013/benchmarks.
The last part of the figure compares the runtime performance to lines of code ratio. Js-scala shows the best score, being 1.48 times better than the vanilla JavaScript version, 1.88 times better than the HaXe version, 3.45 times better than the GWT version and 7.82 times better than the jQuery JavaScript version.

5.2.2 Code Reuse

We were able to share some DOM fragment definitions between server-side and client-side only in the js-scala version. In the GWT version we don’t have a choice: dynamic DOM fragments are always built only on client-side (a practice that makes it more difficult to make the pages content crawlable by search engines and may increase the initial display time). In the other versions the code for building the DOM fragment is duplicated between client and server sides, representing 20 lines of JavaScript code (17% of the total) and 15 lines of HTML (5% of the total) in the JavaScript version, and 19 lines of HaXe code (13% of the total) and 15 lines of HTML (5% of the total) in the HaXe version. In the js-scala version the DOM fragment definitions shared between clients and servers represent 22 lines of Scala code (30% of the total) and save 15 lines of HTML (5% of the total).

5.2.3 Threats to Validity

Our goal was to put the runtime performance in perspective with the level of abstraction. We are aware that the indicator we chose as an inverse approximation of the abstraction level, the number of lines of code, is not scientifically established and may be subject to discussion. However, we think it is a reasonable approximation in our case because all the candidate languages we use have a similar syntax, inherited from the C programming language.

Another weakness of our validation may come from the fact that our application does not make a heavy use of client-side code and thus may not be representative of the way large Web applications are written. However, we think that a richer application would have more parts of code susceptible to be shared between client and server sides, thus giving even better results on the code reuse statistics.

Finally, the GWT version may not have been written in an as idiomatic as possible way. Indeed, we mainly catch the events directly on the HTML DOM, as we do in JavaScript, without reusing all the GWT widgets. We do not build the application as a blank page with a set of widgets. However, this way of developing using GWT has no impact on the performance and a minor impact on the number of lines of code.

6. Discussion

We implemented our solution as compiled embedded DSLs in Scala. Generating code from our DSLs is a two step process: an initial Scala program first evaluates to an intermediate representation of the final program that is traversed by code generators to produce the final JavaScript code. Domain-specific optimizations
can happen during the IR construction (as shown in section 4.1) or during the code generation (as shown in section 4.2).

An important consequence of the implementation as compiled embedded DSLs is that defining a DSL that can be shared between server and client sides requires a low effort: compiled embedded DSLs are simply defined as libraries but let developers specialize the generated code according to each target platform (as shown in section 4.3).

In other words, the compiled embedded DSL approach gives us a way to exploit the Scala host language to define high-level language units that integrate seamlessly together and bring domain-specific knowledge to the code generation scheme to produce efficient code for client and server sides.

These characteristics allowed us to capture some Web programming patterns as high-level abstractions, making the code of our application simpler to reason about and making some parts of the code reusable between client and server sides, while keeping execution performance on client-side as high as if we used hand-tuned low-level JavaScript code.

7. Conclusion

High-level abstractions for Web programming, which are useful to decrease the complexity of the code and to abstract over the differences between the client and server environments, must be implemented in a way to efficiently run on hardware with limited capabilities.

In this paper we showed how to leverage staging to implement high-level abstractions for Web programming that are efficiently compiled for heterogeneous platforms such as Web clients and servers that differ in their technical API. We also showed how these abstractions can be shared between client and server sides.

Our two kinds of benchmarks, (i) micro-benchmarks to evaluate one abstraction and (ii) a benchmark on a real application using these abstractions, show performance similar to hand-optimized low-level code.

In a future work we may investigate more coarse-grained optimizations like smart DOM updates minimizing the number of browser reflows.

References