Efficient High-Level Abstractions for Web Programming

Richard-Foy, Olivier Barais, and 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.

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 [1,2]. One challenge comes from the fact that the business logic is scattered into heterogeneous client-side and server-side environments [3,4]. 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 [5,6].

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 [7], SharpKit[1], Dart [8], Kotlin[2], ClojureScript [9], Fay[3], Haxe [10] or Opa[4]).

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) [11,12]. 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-

[1] http://sharpkit.net
[2] http://kotlin.jetbrains.org/
[3] http://fay-lang.org/
[4] http://opalang.org/

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[5].

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.

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 [14], Opa, Dart [8].

Domain-Specific Languages Another approach consists in defining several independent domain-specific languages [15], 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. [16] gave an example of such a domain-specific language for defining Web applications.

[5] The code is available at http://github.com/js-scala

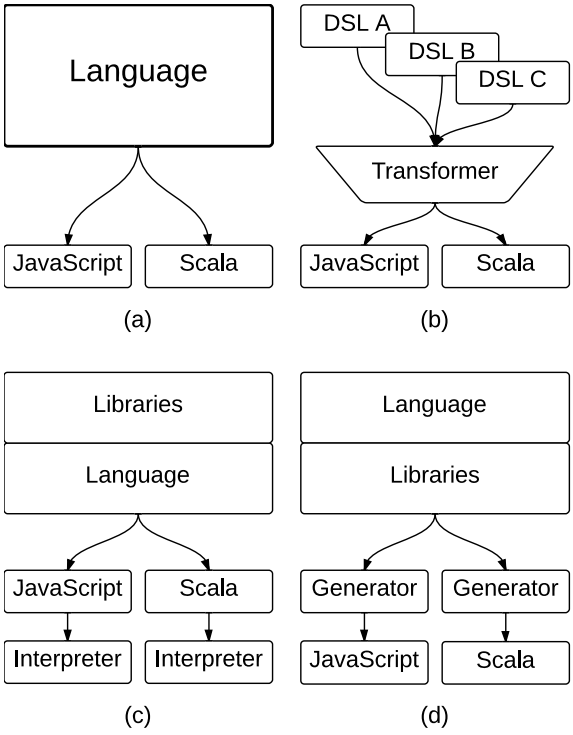

Fig. 1. Language engineering processes

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 [17]) 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 [18,19] (LMS) is a framework for defining deeply embedded DSLs in Scala. It has been used to define high-performance DSLs for parallel computing [20] and to define JavaScript as an embedded DSL in Scala [13].

LMS is based on staging [21]: 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.

3.1 Walkthrough: defining a DSL component

For readers who are new to LMS, this section shows how to implement a language unit for logging.

We start by defining the language unit vocabulary in an abstract trait:

```
trait Debug extends Base {
  def log(msg: Rep[String]): Rep[Unit]
}
```

The `Base` trait is part of the core LMS framework and provides the abstract type constructor `Rep`.

The trait contains just one method, `log`, representing the action of appending a message to the log.

Then we define the language unit implementation:

```
trait DebugExp extends Debug with EffectExp {
  case class Log(msg: Exp[String]) extends Def[Unit]
  def log(msg: Exp[String]): Exp[Unit] = reflectEffect(Log(msg))
}
```

The `EffectExp` trait is part of the core LMS framework. It inherits from `BaseExp` which aliases `Rep` to `Exp`. `Exp` represents an IR via two subclasses: `Const` for constants and `Sym` for symbols referring to a composite expression. Composite expressions are represented by type `Def`.

In our `DebugExp` trait, we extend `Def` to define a new IR node: `Log`. This node represents a statement appending a message to the log.

An LMS program returns a value of type `Rep[_]`, a graph of IR nodes, representing the staged program. This graph is traversed and a code generator is invoked to transform each node into the corresponding code in the target language. To implement code generation to JavaScript for our logging IR node, we simply override `emitNode` to handle `Log`:

```
trait JSGenDebug extends JSGenEffect {
  val IR: DebugExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case Log(s) => emitValDef(sym, q"console.log($s)")
      case _ => super.emitNode(sym, rhs)
    }
}
```

We chose to implement the logging action by using the standard `console.log` Web browser's function.

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 a HTML document

according to their name or attribute values. Figure 2 summarizes the available functions and their differences.

Function	Description
`querySelector(s)`	First element matching the CSS selector s
`getElementById(i)`	Element which attribute id equals to i
`querySelectorAll(s)`	All elements matching the CSS selector s
`getElementsByTagName(n)`	All elements of type n
`getElementsByClassName(c)`	All elements which **class** attribute contains c

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

```
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]
      .getElementsByClassName('word');
    results[i] = words;
  }
  return results
}
```

Listing 1.1. Searching elements using the native selectors API

Listing 1.1 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 its `fieldset` 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.

```
function getWords() {
  var form = $('#add-user');
  var sections = $('fieldset', form);
  return sections.map(function () {
    return $('.word', this)
  })
}
```

Listing 1.2. Searching elements using jQuery

A high-level abstraction for searching elements in a document could be just one function finding all elements matching a given CSS selector. In fact, most JavaScript developers[6] use the jQuery library that actually provides only one function to search for elements. Listing 1.2 shows an equivalent JavaScript program as Listing 1.1, 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 =>
      DocumentGetElementById(Const(id))
    case _ =>
      SelectorFind(receiver, selector)
  }
```

Listing 1.3. Selectors optimization

Our API has two functions: `find` to find the first element matching a selector and `findAll` to find all the matching elements. Listing 1.3 gives the implementation of the `find` function. It is a Scala function that returns an IR

[6] According to http://trends.builtwith.com/javascript, jQuery is used by more than 40% of the top million sites.

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 `DocumentGetElementById` 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 a `SelectorGetElementsByClassName` in case of a CSS class name selector, a `SelectorGetElementsByTagName` in case of a CSS tag name selector, and a `SelectorFindAll` otherwise.

```
def getWords() = {
  val form = document.find("#add-user")
  val sections = form.findAll("fieldset")
  sections map (_.findAll(".word"))
}
```

Listing 1.4. Searching elements in js-scala

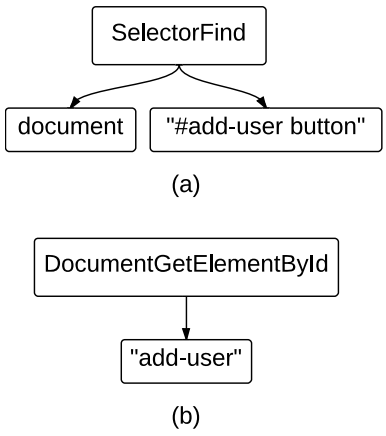

Fig. 3. Intermediate representations returned by the evaluation of (a) `document.find("#add-user button")` and (b) `document.find("#add-user")`

Figure 3 shows the IRs returned by the evaluation of *document .find("#add − userbutton")* and *document.find("#add − user")*.

In the former case, the selector is parsed and does not match an ID selector (it is a composite selector matching button elements within the element having the add−user id), so a SelectorFind node is returned, then translated into a call to the general querySelector function. In the latter case, the selector matches an ID selector so a DocumentGetElementById node is returned, then translated into a call to the specialized getElementById function.

Finally, Listing 1.4 shows how to implement Listing 1.2 in Scala using js-scala. The code has the same abstraction level as with jQuery, however it generates a JavaScript program identical to Listing 1.1: the high-level abstractions (the find and findAll functions) exist only in the initial program, not in the final JavaScript program.

4.2 Monads Sequencing

This section presents an abstraction to handle **null** references and shows how this abstraction can be shared between client and server code.

null references are a known source of problems in programming languages [22,23]. For example, consider Listing 1.5 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 1.6.[7]

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

Listing 1.5. Unsafe code

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.

[7] However, one could alleviate the syntax burden by using a language such as CoffeeScript [24], that suppports a special notation for optional values dereferencing and desugars directly to JavaScript.

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

Listing 1.6. Defensive programming to handle null references

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 1.7, that implements in js-scala a program equivalent to Listing 1.6. 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.

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

Listing 1.7. 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 1.7 produces a code equivalent to Listing 1.6.

Listing 1.8 shows the JavaScript code generator for methods `isEmpty` (that checks if the optional value contains a value) and `foreach` (that is called when the **for** notation is used, as in Listing 1.7). The `emitNode` method handles `OptionIsEmpty` and `OptionForeach` nodes returned by the implementations of `isEmpty` and `foreach`, respectively. In the case of the `OptionIsEmpty` node, it simply generates an expression testing 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**.

```
override def emitNode(sym: Sym[Any], rhs: Def[Any]) =
  rhs match {
    case OptionIsEmpty(o) =>
      emitValDef(sym, q" $o === null")
    case OptionForeach(o, b) =>
      stream.println(q"if ($o !== null) {")
      emitBlock(b)
      stream.println("}")
    case _ =>
      super.emitNode(sym, rhs)
  }
```

Listing 1.8. JavaScript code generator for null references handling DSL

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 JavaScript 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.

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

Listing 1.9. JavaScript DOM creation native 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

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

Listing 1.10. Scala XML API

the server-side (to initially respond to a request) or from the client-side (to update the current page). As an example, Listing 1.9 defines a JavaScript function `articleUi` that builds a DOM tree containing an article description. Listing 1.10 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 1.11 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 1.12.

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

Listing 1.11. DOM definition DSL

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

Listing 1.12. Using loops

The el function returns an Element IR node that is a tree composed 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 1.11) rather than dynamically computed (as in Listing 1.12), the code generators unroll the loop that adds children to their parent, for better performance. As a result, Listing 1.11 generates a code equivalent to Listing 1.9 on client-side and equivalent to Listing 1.10 on server-side.

```
case Tag(name, children, attrs) =>
  emitValDef(sym, q"document.createElement('$name')")
  for ((n, v) <- attrs) {
    stream.println(q"$sym.setAttribute('$n', $v);")
  }
  children match {
    case Left(children) =>
      for (child <- children) {
        stream.println(q"$sym.appendChild($child);")
      }
    case Right(children) =>
      val x = fresh[Int]
      stream.println(q"for (var $x = 0; $x < $children.length; $x++) {")
      stream.println(q"$sym.appendChild($children[$x]);")
      stream.println("}")
  }
case Text(content) =>
  emitValDef(sym, q"document.createTextNode($content)")
```

Listing 1.13. JavaScript code generator for the DOM fragment definition DSL

Listings 1.13 and 1.14 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 approaches. We take the number of lines of code as an inverse approximation 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 benchmarked 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 application in an idiomatic way). The performance benchmarks measured the

```
case Tag(name, children, attrs) =>
  val attrsFormatted =
    (for ((name, value) <- attrs)
     yield q" $name={ $value }").mkString
  children match {
    case Left(children) =>
      if (children.isEmpty) {
        emitValDef(sym, q"<$name$attrsFormatted />")
      } else {
        emitValDef(sym,
          q"<name$attrsFormatted>{ ${children.map(quote)} }</$name>"
        )
      }
    case Right(children) =>
      emitValDef(sym, q"<$name$attrsFormatted>{ $children }</$name>")
  }
case Text(content) =>
  emitValDef(sym, q"{xml.Text(content)}")
```

Listing 1.14. Scala code generator for the DOM fragment definition DSL

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[8].

Selectors 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: the best we could do in GWT or Haxe is to expose the native high-level API (`querySelector` and `querySelectorAll`). So we directly compared the execution time of the JavaScript code generated by Listing 1.4 with a JavaScript program equivalent to Listing 1.1 but using the high-level native API (`querySelector` and `querySelectorAll`) instead. The code was executed in a Web page containing a few elements: 4 `fieldset` elements, each containing 0 to 2 elements with class `word`.

[8] The source code of the benchmarks is available at https://github.com/js-scala/js-scala/tree/master/papers/gpce2013/benchmarks

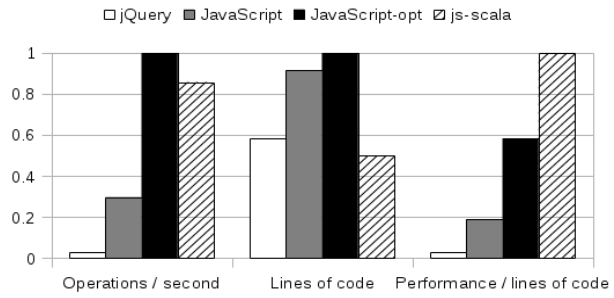

Fig. 4. Micro-benchmark on the selectors abstraction

Figure 4 shows the benchmark results. The JavaScript-opt version is Listing 1.1, 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 1.2. 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.

Optional Value We reimplemented the optional value abstraction in plain JavaScript, Java and HaXe and wrote a small program manipulating optional values. Listing 1.15 shows the js-scala version of this program. The `maybe` function is a function partially defined on `Int` values.

```
val maybe = fun { (x: Rep[Int]) =>
  some(x + 1)
}

def benchmark = for {
  a <- maybe(0)
  b <- maybe(a)
  c <- maybe(b)
  d <- maybe(c)
} yield d
```

Listing 1.15. Micro-benchmark code for the optional values abstraction

Figure 5 shows the benchmark results. The js-scala version of the program runs between 3 to 10 times faster than other approaches. This version also takes less lines of code than others (this result is almost due to the special **for** notation, that has no equivalent in other benchmarked languages). Finally, the js-scala program has a performance to lines of code ratio more than 4 times higher than others.

Fig. 5. Micro-benchmark on the optional values abstraction

5.2 Real World Application

Chooze [9] is an existing complete application for making polls. It allows users to create a poll, define the choice alternatives, share the poll, vote and look at the results. It contains JavaScript code to handle the dynamic behavior of the application: double-posting prevention, dynamic form update and rich interaction with the document. The size of the whole application (server and client sides) is about one thousand lines of code.

The application was initially written using jQuery. We rewrote it in vanilla JavaScript (low-level hand-tuned code without third-party library), js-scala, GWT and HaXe.

Performance The benchmark code simulates user actions on a Web page (2000 clicks on buttons, triggering a dynamic update of the page and involving the use of the optional value monad, the selectors API and the HTML fragment definition API). Figure 6 shows the benchmark results.

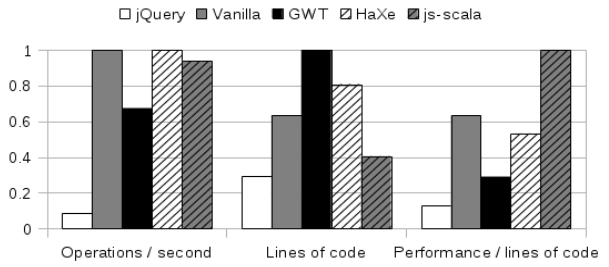

Fig. 6. Benchmarks on a real application

[9] Source code is available at http://github.com/julienrf/chooze, under the branches `vanilla`, `jquery`, `gwt`, `haxe` and `js-scala`

The runtime performance of the vanilla JavaScript, HaXe and js-scala versions are similar (though the js-scala version is slightly slower by 6%). It is worth noting that the vanilla JavaScript and the HaXe versions use low-level code compared to js-scala, as shown in the middle of the figure (lines of code): the js-scala version needs only 74 lines of code while the vanilla JavaScript version needs 116 lines of code (57% bigger) and the HaXe version needs 148 lines of code (100% bigger). The jQuery JavaScript version, which code is high-level (54 lines of code, 27% less than js-scala) runs 10 times slower than the js-scala version.

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.

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).

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

1. Mikkonen, T., Taivalsaari, A.: Web applications - spaghetti code for the 21st century. In: Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications, Washington, DC, USA, IEEE Computer Society (2008) 319–328

2. Preciado, J.C., Trigueros, M.L., Sánchez-Figueroa, F., Comai, S.: Necessity of methodologies to model rich internet applications. In: WSE, IEEE Computer Society (2005) 7–13
3. Rodríguez-Echeverría, R.: Ria: more than a nice face. In: Proceedings of the Doctolral Consortium of the International Conference on Web Engineering. Volume 484., CEUR-WS.org (2009)
4. Kuuskeri, J., Mikkonen, T.: Partitioning web applications between the server and the client. In: Proceedings of the 2009 ACM symposium on Applied Computing. SAC '09, New York, NY, USA, ACM (2009) 647–652
5. Souders, S.: High-performance web sites. Communications of the ACM **51**(12) (2008) 36–41
6. Huang, J., Xu, Q., Tiwana, B., Mao, Z.M., Zhang, M., Bahl, P.: Anatomizing application performance differences on smartphones. In: Proceedings of the 8th international conference on Mobile systems, applications, and services, ACM (2010) 165–178
7. Chaganti, P.: Google Web Toolkit: GWT Java Ajax Programming. Packt Pub Limited (2007)
8. Griffith, R.: The dart programming language for non-programmers-overview. (2011)
9. McGranaghan, M.: Clojurescript: Functional programming for javascript platforms. Internet Computing, IEEE **15**(6) (2011) 97–102
10. Cannasse, N.: Using haxe. The Essential Guide to Open Source Flash Development (2008) 227–244
11. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys **28** (1996)
12. Elliott, C., Finne, S., De Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(3) (2003) 455–481
13. Kossakowski, G., Amin, N., Rompf, T., Odersky, M.: JavaScript as an Embedded DSL. In Noble, J., ed.: ECOOP 2012 – Object-Oriented Programming. Volume 7313 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 409–434
14. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Formal Methods for Components and Objects, Springer (2007) 266–296
15. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM Sigplan Notices **35**(6) (2000) 26–36
16. Visser, E.: WebDSL: A case study in domain-specific language engineering. In Lämmel, R., Visser, J., Saraiva, J., eds.: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007. Volume 5235 of Lecture Notes in Computer Science., Braga, Portugal, Springer (2007) 291–373
17. Bibeault, B., Kats, Y.: jQuery in Action. Dreamtech Press (2008)
18. Rompf, T.: Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE (2012)
19. Rompf, T., Sujeeth, A., Amin, N., Brown, K., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., Odersky, M.: Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers based on Staging. Technical report (2012)
20. Brown, K.J., Sujeeth, A.K., Lee, H.J., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In:

Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, IEEE (2011) 89–100
21. Jørring, U., Scherlis, W.L.: Compilers and staging transformations. In: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM (1986) 86–96
22. Hoare, T.: Null references: The billion dollar mistake. Presentation at QCon London (2009)
23. Nanda, M., Sinha, S.: Accurate interprocedural null-dereference analysis for java. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, IEEE (2009) 133–143
24. Ashkenas, J.: Coffeescript (2011)