How to build an homogeneous language for heterogeneous platforms

You don't have to trade abstraction for control

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 little 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 engineering comfort for performance? This article presents high-level abstractions defined as deep embedded DSLs in Scala, that can (1) generate efficient code leveraging the target platform characteristics, (2) be shared between client and server code. Code written with our DSL has a high level of abstraction but runs as fast as hand tuned low-level JavaScript code.

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

General Terms Languages, Software Engineering

Keywords Heterogeneous code generation, Domain-specific languages

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 [14, 16]. One challenge comes from the fact that the business logic is scattered into heterogeneous client-side and server-side environments [12, 17]. This gives less flexibility in the engineering process and requires a higher maintenance effort: if a piece of logic is implemented on client-side and finally needs to be implemented on server-side instead, the code can not be reused and the feature needs to be completely rewritten (and *vice versa*). 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 an 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 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 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. "Every 100 ms delay costs 1% of sales", said Amazon in 2006.

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 the action language on Web clients – can be used on server-side, and an increasing number of programming languages or compiler back-ends can generate JavaScript code (*e.g.*Java/GWT [4], SharpKit[1], Dart [7], Kotlin[2], ClojureScript [13], Fay[3], Haxe [3] 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 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.

[1] http://sharpkit.net

[2] http://kotlin.jetbrains.org/

[3] http://fay-lang.org/

[4] http://opalang.org/

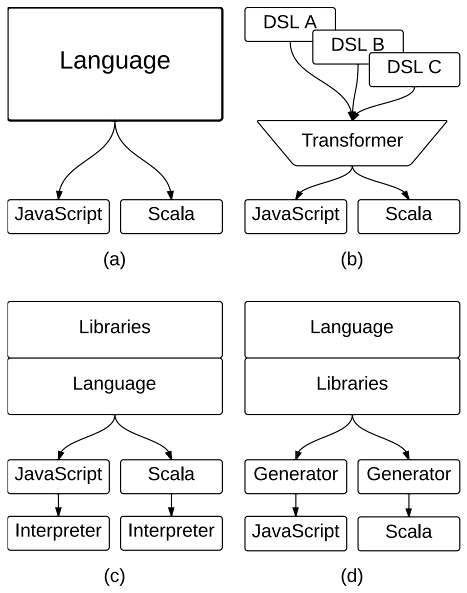

Figure 1. Language engineering processes

But on the other hand, for performance reasons, they want to keep control on the way their code is compiled to each target platform. How to solve this dilemma?

Compiled domain-specific embedded languages [6] allow the definition of domain-specific languages (DSLs) as libraries on top of a host language, and to compile them to a target platform. The 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 JavaScript [11]. However, Kossakowski *et al.* did not address the engineering dilemma described above. This paper enriches *js-scala* with the following contributions:

- we define high-level abstractions typically used in Web programming, that generate low-level code leveraging native Web browsers APIs;
- in the case of abstractions shared between servers and clients, we specialize the code generation in order to leverage the target platform native environments.

We validate our approach with a case study implemented with various candidate technologies and discuss the relative pro and cons of them. Though the code written in our DSL is high-level and can be shared between clients and servers, it has the same runtime performances as hand-tuned low-level JavaScript code.

The remainder of this paper is organized as follows. The next section introduces existing approaches for defining cross-compiling languages. Sections 3 and 4 present our contribution. Section 5 gives implementation details. Section 6 evaluates the contribution. Section 7 concludes.

2. Background

This section presents different approaches for defining cross-platform programming languages.

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, and more specific concepts such as HTML fragment definition. Examples of such languages are Links [5], Opa and Dart [7]. These languages are difficult to extend because each concept is defined in the compiler, and modifying a compiler requires a high effort. Furthermore, these languages also require to support common programming abstraction and composition mechanisms, as general purpose languages do. So they usually try to re-invent the features of general purpose languages. We argue that this approach for defining programming languages is difficult to scale: for every problem you have to rewrite a full-featured programming language besides addressing the concepts specific to the problem domain.

Domain-Specific Languages Another approach consists in defining several independent domain-specific languages [19], 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 a language requires a minimal effort compared to the previous approach. On the other hand, it is difficult to have interoperability between DSLs (ref). [8] shows such a domain-specific language for defining Web applications.

Thin Languages Alternatively, one can define concepts relative to a specific domain as libraries on top of a thin general purpose language (this is also referred to as *embedded* domain specific languages). Figure 1 (c) depicts this approach. The general purpose language is used as a host language and does not need to be modified if a new concept is introduced, because concepts are defined as pure libraries. However, this approach gives no opportunity to translate a concept efficiently according to the target platform characteristics. Examples of languages following this approach are Java/GWT, Kotlin, HaXe and SharpKit.

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. (benefits and limitations of this approach)

Lightweight Modular Staging [18] (LMS) is a framework for defining deeply embedded DSLs in Scala using staging [10]. It has been used to define high-performance DSLs for parallel computing [2]. (Todo. Mention `Rep[_]`)

3. High-Level Abstraction for Client-Side Code: Selectors

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 a mouse click). 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.

The `querySelector` and `querySelectorAll` are the most general functions while the others handle special cases. For the developer it is not convenient to have to master several functions performing similar tasks. In fact, most JavaScript developers use

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`

Figure 2. Standard selectors API

Listing 1. Selectors in plain JavaScript
```
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 2. Selectors in jQuery
```
function getWords() {
  var form = $('#add-user');
  var sections = $('fieldset', form);
  return sections.map(function () {
    return $('.word', this)
  })
}
```

Listing 3. Selectors in js-scala
```
def getWords() = {
  val form = document.find("#add-user")
  val sections = form.findAll("fieldset")
  sections map (_.findAll(".word"))
}
```

the jQuery library [1][5] that provides only one high-level function to search for elements. Listings 1 and 2 show two equivalent JavaScript programs performing element searches, the first one using the native APIs and the second one using jQuery.

jQuery provides an API that is simpler to master because it has less functions, but by doing so it can not benefit from the performance of the browser's implementation of specialized search functions (`getElementById`, `getElementsByTagName` and `getElementsByClassName`).

Listing 3 shows how to implement listing 2 using *js-scala*. We provide two functions for searching elements: `find` to find the first element matching a selector and `findAll` to find all the matching elements. During staging these functions try to analyze the selector that is passed as parameter and, when appropriate, produce code using the specialized API, otherwise they produce code using `querySelector` and `querySelectorAll`. As a result, listing 3

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

Listing 4. Unsafe code
```
var loginWidget =
  document.querySelector("div.login");
var loginButton =
  loginWidget.querySelector("button.submit");
loginButton.addEventListener("click", handler);
```

Listing 5. Defensive programming to handle null references
```
var loginWidget =
  document.querySelector("div.login");
if (loginWidget !== null) {
  var loginButton =
    loginWidget.querySelector("button.submit");
  if (loginButton !== null) {
    loginButton.
      addEventListener("click", handler);
  }
}
```

Listing 6. Handling null references in js-scala
```
for {
  loginWidget <- document.find("div.login")
  loginButton <- loginWidget.find("submit.button")
} loginButton.on(Click)(handler)
```

generates a JavaScript program identical to listing 1: the high-level abstraction (the `find` and `findAll` functions) exist only in the initial program, not in the final JavaScript program. It gives two advantages: (1) the execution of the final program is more performant because of the use of specialized APIs, (2) the final program's size is smaller because it does not need to include jQuery. (Move this last sentence in the discussion)

4. High-Level Abstractions Shared on Clients and Servers

4.1 Monads Sequencing

Null references are a known source of problems in programming languages [9, 15]. For example, consider listing 4 finding a particular widget in the page and then a particular button in 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 5.

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.

Listing 6 implements in *js-scala* a program equivalent to listing 5. The `for` notation is used to sequence computations over optional values encoded with a monad. The `find` function returns a `Rep[Option[Element]]` value, that can either be a `Rep[Some[Element]]` (if an element was found) or a `Rep[None.type]` (if no element was found). 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.

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.

Listing 7. JavaScript DOM API

```
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 8. Scala XML API

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

In our case, we expose the monadic API at the first stage but we generate code that does not wrap values in a container object but instead checks if they are `null` or not when dereferenced. So the extra container object exists only at the stage-level and is removed during code generation: listing 6 produces a code equivalent to listing 5.

Last but not least, we leverage the modular architecture of LMS to define two code generators for this API: one targeting JavaScript and the other targeting Scala. So the same abstraction is efficiently translated on both server and client sides.

4.2 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.

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 7 defines a JavaScript function `articleUi` that builds a DOM tree containing an article description, and listing 8 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 API are very different and that the client-side API is very low-level and inconvenient to use.

Instead, we provide a common high-level DSL for defining HTML fragments and we make this DSL generate code leveraging native environments. Listing 9 shows how to implement our example with our DSL. The `el` function defines 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. The children elements of an element can also be obtained dynamically from a collection, as shown in listing 10.

The `el` function returns an `Element` IR node that is a tree composed of other `Element` nodes and `Text` nodes. This tree is traversed by the code generators to produce code building an equivalent DOM tree on client-side and code building an equivalent XML fragment on server-side. When the children elements of an element are constant values (as in listing 9) rather than dynamically obtained (as in listing 10), the code generators inline the loop that adds children to their parent, for better performance. As a result,

Listing 9. DOM definition DSL

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

Listing 10. Using loops

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

Listing 11. Selectors optimization

```
def find(receiver: Exp[Selector],
         selector: Exp[String]) =
  getConstIdentifier(selector) match {
    case Some(id) if receiver == document =>
      DocumentGetElementById(Const(id))
    case _ =>
      SelectorFind(receiver, selector)
  }
```

listing 9 generates a code equivalent to listing 7 on client-side and equivalent to 8 on server-side.

5. Implementation

[6]

5.1 Selectors

Listing 11

5.2 Null references

Code generation consists in traversing the statement nodes produced by the program evaluation according to their dependencies and to emit the code corresponding to each statement. LMS already sorts the statements graph so DSL authors just need to say how to emit code for each statement node of their DSL. Listing 12 shows such a code generator for the `null` reference handling DSL. The `emitNode` method handles `OptionIsEmpty` and `OptionForeach` nodes. 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`.

5.3 DOM Fragments

6. Evaluation

We implemented several applications using js-scala. We also have written several implementations of a complete application, using different approaches to write the client and server sides, and compared the amount of code written, the runtime performances and the ability to modularize the code and to maintain it.

6.1 Real World Application

Chooze [7] 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

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

[7] http://chooze.herokuapp.com

Listing 12. Null reference handling DSL code generator

```
trait JSGenOptionOps extends JSGenEffect {
  val IR: OptionOpsExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any]) =
    rhs match {
      case OptionIsEmpty(o) =>
        emitValDef(sym, q" $o === null")
      case OptionForeach(o, a, b) =>
        stream.println(q"if ($o !== null) {")
        emitValDef(a, quote(o))
        emitBlock(b)
        stream.println("}")
      case _ =>
        super.emitNode(sym, rhs)
    }
}
```

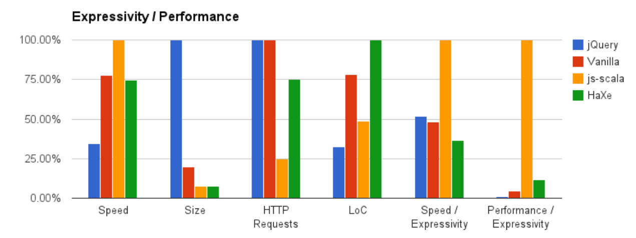

Figure 3. Benchmarks on a real application

to handle the dynamic behavior of the application: double-posting prevention, dynamic form update and rich interaction with the document.

The application was initially written using jQuery. We rewrote it using several technologies for the client-side part: plain JavaScript (without third-party library), js-scala, GWT and HaXe.

Vanilla JavaScript Low-level code.

jQuery High-level code.

js-scala High-level code. HTML fragment definition reused between server and client sides.

HaXe Low-level code.

GWT High-level code?

6.2 Benchmarks, Code Metrics

Our goal is to evalute the level of abstraction provided by each solution and their performances. We took the number of lines of code as a measure of the level of abstraction. We also measured the size of the data sent to the client-side. We also measured the ability to share code between client and server sides.

Rather than doing a micro-benchmark focusing on just one abstraction, we performed a global benchmark that is more likely to reflect the real performances of the application: we simulated user actions on a Web page (2000 clicks on buttons, triggering a dynamic update of the page and involving the use of the Option monad, the selectors API and the HTML fragment definition API) and measured the time it took complete them. We only measured the execution time of client-side code execution. The tests were run on a DELL Latitude E6430 laptop with 8 GB of RAM, on the Google Chrome v27 Web browser. Figure 3 shows the benchmark results.

7. Conclusion

We implemented a high-level language abstracting over client and server heterogeneity but producing efficient code.

Acknowledgments

Acknowledgments, if needed.

References

[1] B. Bibeault and Y. Kats. *jQuery in Action*. Dreamtech Press, 2008.

[2] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100. IEEE, 2011.

[3] N. Cannasse. Using haxe. *The Essential Guide to Open Source Flash Development*, pages 227–244, 2008.

[4] P. Chaganti. *Google Web Toolkit: GWT Java Ajax Programming*. Packt Pub Limited, 2007.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[6] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[7] R. Griffith. The dart programming language for non-programmers-overview. 2011.

[8] D. Groenewegen, Z. Hemel, L. Kats, and E. Visser. Webdsl: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 779–780. ACM, 2008.

[9] T. Hoare. Null references: The billion dollar mistake. *Presentation at QCon London*, 2009.

[10] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 86–96. ACM, 1986.

[11] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript as an Embedded DSL. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 409–434, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-31057-7_19. URL `https://github.com/js-scala/js-scala/`.

[12] J. Kuuskeri and T. Mikkonen. Partitioning web applications between the server and the client. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 647–652, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. doi: 10.1145/1529282.1529416. URL `http://doi.acm.org/10.1145/1529282.1529416`.

[13] M. McGranaghan. Clojurescript: Functional programming for javascript platforms. *Internet Computing, IEEE*, 15(6):97–102, 2011.

[14] T. Mikkonen and A. Taivalsaari. Web applications - spaghetti code for the 21st century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 319–328, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3302-5. doi: 10.1109/SERA.2008.16. URL `http://dl.acm.org/citation.cfm?id=1443226.1444030`.

[15] M. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 133–143. IEEE, 2009.

[16] J. C. Preciado, M. L. Trigueros, F. Sánchez-Figueroa, and S. Comai. Necessity of methodologies to model rich internet applications. In *WSE*, pages 7–13. IEEE Computer Society, 2005. ISBN 0-7695-2470-2.

[17] R. Rodríguez-Echeverría. 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.

[18] T. Rompf. *Lightweight Modular Staging and Embedded Compilers*. PhD thesis, IC, Lausanne, 2012. URL `http://library.epfl.ch/theses/?nr=5456`.

[19] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.