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 code partitioned between heterogeneous client-side and server-side makes the engineering process of Web applications tedious. We use a deep embedded DSL to define a common language abstracting over platform differences for the developers point of view, but generating efficient code leveraging the target platforms characteristics and giving them control on the amount of code produced on client-side.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

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 [12, 16]. One challenge comes from the fact that the business logic is scattered into heterogeneous client-side and server-side environments [10, 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 (explain why?), and the higher is the amount of client-side code. Developers use libraries and frameworks to get higher-level abstractions on client-side, making their code easier to reason about and to main-

tain. Unfortunately, JavaScript – which is currently the most supported the action language on Web clients – has no static linking system allowing to pick just the functions actually used by a program, forcing clients to download the whole library even though only a few functions are actually used.

Using the same programming language on both server-side and client-side can improve the software engineering process by enabling code reuse between both sides. Incidentally, the JavaScript language 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 [11], 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?

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.

For instance, because the boundaries of the code sent to the client are less visible when you share code between client-side and server-side, transitive dependencies may pull a lot of code on the client, causing a high download overhead. Moreover, 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 for performance reasons).

On one hand, for engineering reasons, developers want to write Web applications using a single language, abstracting over the target platforms differences. 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
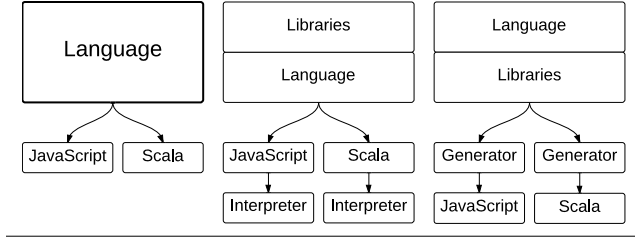
---

[1] http://sharpkit.net

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

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

[4] http://opalang.org/

Figure 1. Language engineering processes

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

Figure 2. Standard selectors API

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.

`js-scala` [9] is such 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. This paper enriches js-scala in order to solve the problems described above. We use staging (explain) to generate efficient code for typical abstractions used for Web programming and to generate efficient specialized code for both client and server sides for shared abstractions. More precisely, we demonstrate the following features:

- An API for searching in the DOM, that exposes a single entry point but that generates code potentially using more optimized native APIs;

- Usage of monads without extra container object creation;

- Ability to define DOM fragments using a common language for server-side and client-side, but that generates code using standard APIs on both server-side and client-side;

- Type-directed ad-hoc polymorphism on client-side without run-time dynamic dispatch logic;

- Static linking.

Compared to libraries offering the same level of abstraction, we observe that js-scala produces significantly smaller executables with better runtime performances.

The remainder of this paper is organized as follows. The next section introduces existing approaches for defining cross-compiling languages. Section 3 presents our contribution. Section 5 evaluates our contribution. Section 6 concludes.

2. Related Work

Developers express programs using a language that is translated into a form executable on a target platform. This section presents different approaches for defining cross-platform languages.

2.1 Fat Languages

One way consists in hard-coding, in the language's compiler, the transformation to each target platform. Figure 1 (left) 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. 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 abstraction and composition mechanisms, as general purpose languages do, in order to make it easier to write real programs. So they need to be at least general purpose languages. It means that for every problem you have to rewrite a full-featured programming language before addressing the concepts specific to the problem domain. This approach does not scale.

2.2 Domain Specific Languages

Problem: interoperability.

2.3 Thin Languages

Alternatively, one can define concepts relative to a specific domain as libraries on top of a thin general purpose language. Figure 1 (middle) depicts this approach. The general purpose language is used as a host language that does not need to be modified if a new concept is introduced, because concepts are defined as libraries. However, this approach gives no opportunity to translate a concept efficiently, according to the host platform characteristics.

2.4 Deeply Embedded Languages

Figure 1 (right) shows a last approach that tries to get the benefits of the previous two approaches, without their shortcomings.

Lightweight Modular Staging [18] is a framework for defining deeply embedded DSLs in Scala. It has been used to define high-performance DSLs for parallel computing [2] and can be used to generate JavaScript code [9].

3. High-Level Abstractions Generating Efficient (and Heterogeneous) Code

3.1 Selectors

In a Web application, the user interface is defined by a HTML document which can be manipulated by the JavaScript code to be updated. One typical operation consists in searching some "interesting" element in the document, in order to extract its content, replace it or listen to user events fired on this element. The standard API provides several function to search elements in a Web document according to their characteristics such as their attribute values. Figure 3.1 (wrong reference number) summarizes the available functions and their differences.

The `querySelector` and `querySelectorAll` are the most general functions, 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 the jQuery library (ref and stats) that provides just one function to search for elements. Listing 4 shows two equivalent snippets of code, the first one using the native APIs (a) and the second one using jQuery (b).

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`).

Using LMS we are able to provide a simple API (like jQuery) and to generate code using specialized functions (and hence benefit

```
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
}
```

Figure 3. (a)

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

Figure 4. (b)

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

Figure 5. Selectors in Js-Scala

from their performances), by analyzing the staged program. Listing 3.1 shows how to implement listing 4 using js-scala. This listing generates a JavaScript program similar to listing 4 (a).

Writing the program with js-scala instead of using a JavaScript library such as jQuery is interesting because the abstraction for searching elements exist only in the initial program source code, 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.

3.2 Monads Sequencing

As another illustration of the staging mechanism, we present a simple DSL to handle null references. This DSL provides an abstraction at the stage-level that is removed by optimization during the code generation.

Null references are a known source of problems in programming languages [8, 13]. For example, consider the following typical JavaScript code finding a particular widget in the page and then a particular button in the widget:

Listing 1. Unsafe code
```
var loginWidget =
  document.querySelector("div.login");
var loginButton =
  loginWidget.querySelector("button.submit");
loginButton.addEventListener("click", handler);
```

The native `querySelector` method returns `null` if no node matched the given selector in the document. If we run the above code in a page where the widget is not present, it will throw an error and stop further JavaScript execution. We can write defensive code to handle `null` cases, but it leads to very cumbersome code:

Listing 2. Defensive programming to handle null references
```
var loginWidget =
  document.querySelector("div.login");
if (loginWidget !== null) {
  var loginButton =
    loginWidget.querySelector("button.submit");
```

```
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
};

<div class=article>
  <span>French wine: </span>
  <strong>10</strong>
</div>
```

Figure 6. DOM

```
  if (loginButton !== null) {
    loginButton.
      addEventListener("click", handler);
  }
}
```

We want to define a DSL that has both the safety and performance of listing 2 but the expressiveness of listing 1. We can get safety by wrapping potentially null values of type `Rep[A]` in a container of type `Rep[Option[A]]` requiring explicit dereferencing, we can get expressiveness by using the Scala `for` notation for dereferencing, and finally we can get performance by generating code that does not actually wraps values in a container but instead checks if they are `null` or not when dereferenced. The wrapping container exists only at the stage-level and is removed during the code generation. Here is a Scala listing that uses our DSL (implementation details are given in section 4):

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

The evaluation of the above listing produces a graph of statements from which JavaScript code equivalent to listing 2 is generated.

3.3 DOM Fragments

In this section we show how we can define a template engine as an embedded DSL with minimal effort. This template engine is statically typed and able to insert dynamic content in a safe way. It provides a powerful expression language, requires no extra compilation step and can be used on both client-side and server-side.

Because the template engine is defined as en embedded DSL, we can reuse Scala's constructs:

- a function taking some parameters and returning a DOM fragment directly models a template taking parameters and returning a DOM fragment ;
- the type system type-checks template definitions and template calls ;
- the Scala language itself is the expression language ;
- compiling a template is the same as compiling user code.

So the only remaining work consists in defining the DSL vocabulary to define DOM nodes. We provide a `el` function to define a tag and any `String` value is considered to be a text node.

Listing 3 uses our DSL and generates a code equivalent to listing 3.3. The readability has been highly improved: nesting tags is

Listing 3. DOM definition DSL

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

just like nesting code blocks, HTML entities are automatically escaped in text nodes, developers have the full computational power of Scala to inject dynamic data and DOM fragments definitions are written using functions so they compose just as functions compose. These benefits come with no performance loss because the DSL generates code building DOM fragments by using the native JavaScript API.

Our DSL is equivalent to a template engine with Scala as the expression language. Making it usable on both server and client sides was surprisingly as simple as defining another code generator for the DSL, producing Scala code.

For instance, the template written in listing 3 produces the following Scala code usable on server-side:

```
def articleUi(article: Article) = {
    val x0 = <span>{ article.name + ": " }</span>
    val x1 = <strong>{ article.price }</strong>
    val x2 =
        <div class="article">
            {List(x0, x1)}
        </div>
    x2
}
```

We are able to tackle the code sharing issues described in the introduction (more details?) because of the embdedded nature of our DSLs: dynamic content of templates is written using embedded DSLs too, so their translation into JavaScript and Scala is managed by their respective code generators.

3.4 Ad-Hoc Polymorphism

Because of the dynamically typed nature of JavaScript, when calling a function there is no proper way to select a specialized implementation according to the function's parameters types. JavaScript is only able to dispatch according to a method receiver prototype, *e.g.* if one writes `foo.bar()` the JavaScript runtime will look into the prototype of the `foo` object for a property named `bar` and will call it. So, the only way to achieve *ad hoc* polymorphism on JavaScript objects consists in defining the polymorphic function on the prototypes of the objects. However, modifying existing object prototypes is considered bad practice [20]. Another way could consist in manually coding the dispatch logic, by registering supported data types at the beginning of the program execution, as described in section 2.4.3 of [1], but this solution is painful for developers and incurs a performance overhead.

We propose to achieve *ad hoc* polymorphism using typeclasses [14, 15, 19] so that it supports retroactive extension without modifying objects prototypes. The dispatch logic is type-directed and performed by the compiler, so there is no runtime overhead.

Listing 4 demonstrates how to define a polymorphic `listWidget` function that returns a DOM tree containing the representation of a list of items. The `Show[A]` typeclass defines how to produce a DOM tree for a value of type `A`. It is used by the `listWidget` function to get the DOM fragments of the list items. The listing shows how to reuse the same `listWidget` function to show a list of users and a list of articles.

4. Implementation

See http://github.com/js-scala/js-scala.

Listing 4. Ad hoc polymorphism using typeclasses

```
// Interface
case class Show[A](show: Rep[A => Node])

// Polymorphic function
def listWidget[A](items: Rep[List[A]])
        (implicit A: Show[A]): Rep[Node] =
    el("ul")(
        for (item <- items) yield {
            el("li")(A.show(item))
        }
    )

// Type 'User'
type User = Record {
    val name: String
    val age: Int
}
// Implementation of Show for a User
implicit val showUser = Show[User] { user =>
    el("span", "class"->"user")(
        user.name + "(" + user.age + " years)"
    )
}

// Main program
def main(users: Rep[List[User]]) = {
    document.body.append(listWidget(users))
}
```

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

Figure 7. Selectors optimization

4.1 Selectors

4.2 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 5 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`.

4.3 DOM Fragments

4.4 Ad-Hoc Polymorphism

5. 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.

Listing 5. Null reference handling DSL code generator

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

5.1 Real World Application

Chooze [5] 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 some JavaScript code 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 SharpKit.

5.2 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 and we wrote micro-benchmarks to evaluate the runtime performances of a solution. 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.

6. Conclusion, Future Work

Acknowledgments

Acknowledgments, if needed.

References

[1] H. Abelson and G. Sussman. Structure and interpretation of computer programs. 1983.

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

[9] 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/.

[10] 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.

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

[12] 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.

[13] 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.

[14] M. Odersky. Poor man's Type Classes. In *Presentation at the meeting of IFIP WG*, July 2006.

[15] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, Oct. 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869489. URL http://doi.acm.org/10.1145/1932682.1869489.

[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] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283. URL http://doi.acm.org/10.1145/75277.75283.

[20] N. Zakas. *Maintainable JavaScript*. O'Reilly Media, 2012.

---

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