Title Text

You don't have to trade abstraction for control

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

Abstract

This is the text of the abstract.

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 [11, 15]. One challenge comes from the fact that the business logic is scattered into heterogeneous client-side and server-side environments [9, 16]. This gives less flexibility in the engineering process and requires a higher maintainance effort: once you decided to implement a feature on client-side, changing your *mind* means completely rewriting the feature on server-side (and *vice versa*). Even worse, logic parts that need to run on both client-side and server-side are 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?).

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 – which is currently the action language natively supported by most of 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 [6], Kotlin[2], ClojureScript [10], Fay[3], Haxe [3] or Opa[4]).

However, this engineering comfort may come at the price of an inefficient runtime: abstracting over platform differences often means restricting to a subset of common features and losing opportunities to perform platform specific optimizations. 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 smartphone. "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 hardcoded 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 [5] 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.

This paper presents such a compiled embedded DSL allowing developers to write Web applications in a single language which code fragments can be shared between client and server sides, and which is efficiently compiled to each side. More precisely, we demonstrate the following features:

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

[1] http://sharpkit.net

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

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

[4] http://opalang.org/

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

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

2.1 Fat Languages

2.2 Thin Languages

2.3 Deeply Embedded Languages

Lightweight Modular Staging [17] 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 [8].

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

3.1 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 [19]. 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 [13, 14, 18] so that it supports retroactive extension without modifying objects prototypes because it is type-directed: the dispatch happens at compile-time rather than at runtime.

Listing 1 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.

3.2 Monads Sequencing

The previous work on js-scala [8] showed how to make asynchronous programming more convenient by making asynchronous calls looking like synchronous calls (*i.e.* returning a value instead of taking a callback as parameter). This work was based on the Scala continuations compiler plugin. We claim that, although this programming model removes the "callback hell", it can make the code hard to reason about because there is no explicit distinction between synchronous and asynchronous computations.

We propose a DSL that explicitly reflects the asynchronous nature of computations in their types and provides methods turning them into first-class citizens. The DSL is monadic so we can solve the callback hell problem thanks to the Scala `for` notation.

For instance, listing 2 shows how the listing **??** can be re-written using our DSL. The `createVertex` function now returns an asynchronous value instead of taking a callback as parameter. Then, the

Listing 1. Ad hoc polymorphism using typeclasses

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

// Polymorphic function
def listWidget[A : Show](items: Rep[List[A]]): Rep[Node] =
  tag("ul")(
    for (item <- items) yield {
      tag("li")(implicitly[Show[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 =>
  tag("span", "class"->"user")(
    text(user.name + "(" + user.age + " years)")
  )
}

// Type ‘Article‘
type Article = Record { val name: String; val price: Doubl
// Implementation of Show for an Article
implicit val showArticle = Show[Article] { article =>
  tag("span")(
    text(article.name),
    tag("strong")(text(article.price + " Euros"))
  )
}

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

Listing 2. Asynchronous values are first class citizen

```
def createVertex(text: Rep[String]): Rep[Future[Vertex]] =
  Ajax.post[Vertex]("/create", new Record { val content =

for (vertex <- createVertex("Hello, World!")) {
  addVertex(vertex)
}
```

Listing 3. No callback hell

```
for {
  foo <- Ajax.get(fooUrl())
  bar <- Ajax.get(barUrl(foo))
  _ <- future(println("bar = " + bar))
  baz <- Ajax.get(bazUrl(bar))
} println(foo + bar + baz)
```

`for` expression allows us to get the vertex, when available, and to insert it on the user interface. By making the `createVertex` function return an asynchronous value instead of taking a callback as a parameter, the code is easier to modularize into loosely coupled parts.

Listing 3 translates the callback hell example (listing **??**) using our DSL. The `for` notation can intuitively be thought of as a sequencing notation: whenever the response of the first Ajax request is available, the next statement is executed, and so on. There is no nested callbacks and the order of execution is directly reflected by the order of statements.

Listing 4. Parallel computations in Scala

```
val fooAsync = Ajax.get("/foo")
val barAsync = Ajax.get("/bar")
for {
  foo <- fooAsync
  bar <- barAsync
} println(foo + bar)
```

An asynchronous value of type `Rep[A]` is modelled by a value of type `Rep[Future[A]]`. Because Scala's `for` notation is just syntactic sugar for methods `foreach`, `map` and `flatMap`, we are able to define our DSL by just defining these methods on `Rep[Future[A]]` values, with the following semantic:

- `foreach(f: Rep[A] => Rep[Unit]): Rep[Unit]`, eventually does something with the value when available ;
- `map(f: Rep[A] => Rep[B]): Rep[Future[B]]`, eventually transforms the value when available ;
- `flatMap(f: Rep[A] => Rep[Future[B]]): Rep[Future[B]]` also eventually transforms the value when available ;

Asynchronous values can be used and transformed using the `foreach`, `map` and `flatMap` methods, turning them into first-class citizens: functions can take them as parameters and return them. Consequently, it is easier to write concurrent code: listing 4 shows how to run two parallel asynchronous computations and to do something with their results when both are available. Writing equivalent code in pure JavaScript is tedious because it requires that the callback passed to each asynchronous computation checks if the other has been completed before, and to store the value, when availalbe, in a shared place (so both callbacks can get the other's value).

As an 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 [7, 12]. For example, consider the following typical JavaScript code finding a particular widget in the page and a then particular button in the widget:

Listing 5. Unsafe code

```
var loginWidget = document.querySelector("div.login");
var loginButton = loginWidget.querySelector("button.submit");
loginButton.addEventListener("click", function (e) { ... });
```

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 6. 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", function (e) { ... });
  }
}
```

We want to define a DSL that has both the safety and performance of listing 6 but the expressiveness of listing 5. 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 **??**):

```
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 6 is generated.

Listing 7. DOM definition DSL

```
def vertexDom(v: Rep[Vertex]) =
    tag("g", "class"->"vertex",
            "transform"->("translate("+v.x+","+v.y+")"))(
        tag("rect", "width"->v.width, "height"->v.height)(
        tag("text", "width"->v.width, "height"->v.height)(
            text(v.content)
        )
    )
```

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 typechecks 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 `tag` function to define a tag and a `text` function to define a text node.

Listing 7 uses our DSL and generates a code equivalent to listing **??**. The readability has been highly improved: nesting tags is 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.

3.3.1 Reuse the DOM definition DSL on server-side

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 7 produces the following Scala code usable on server-side (the generated code for client-side is roughly equivalent to listing **??**):

```
def vertexDom(v: Vertex) = {
  val x0 =
    <text width="{v.width}" height="{v.height}">
      {v.content}
    </text>
  val x1 = <rect width="{v.width}" height="{v.height}" />
  val x2 =
    <g class="vertex" transform="translate({v.x},{v.y})">
      {List(x0, x1)}
    </g>
  x2
}
```

We are able to tackle the code sharing issues described in section **??** 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 Selectors

querySelectorAll, getElementsByTagName, getElementsByClassName, getElementById, querySelector

4. Implementation?

5. Evaluation

5.1 Real World Application

Chooze.

5.2 Several Implementations

5.2.1 Vanilla JavaScript

5.2.2 jQuery

5.2.3 GWT

5.2.4 SharpKit

5.2.5 Js-Scala

5.3 Benchmarks, Code Metrics

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

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

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

[8] 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/`.

[9] 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`.

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

[11] 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`.

[12] 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.

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

[14] 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`.

[15] 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.

[16] 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.

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

[18] 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`.

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