Statically Typed Web Programming using Scala

Julien RICHARD-FOY*‡, Olivier BARAIS*, Tiark ROMPF†, and Jean-Marc JÉZÉQUEL*

*IRISA, Université de Rennes 1, France: {first}.{last}@irisa.fr
‡Zenexity, Paris, France: jrf@zenexity.com
†EPFL, Lausanne, Switzerland: tiark.rompf@epfl.ch

Abstract. Rich Internet Applications involve even more code on the client-side, dealing with DOM manipulation, event handling and asynchronous calls to the server. Writing and maintaining large JavaScript code bases is challenging because this language has no static typing and no module system. We investigate further the definition of JavaScript as a Scala compiled embedded DSL using Lightweight Modular Staging. We define safe and expressive abstractions for DOM manipulation, event handling and asynchronous programming that translate to efficient JavaScript code using native APIs. We use both type-level information and staging to introduce high-level abstractions with few or no overhead on the generated JavaScript code.

Keywords: Web, programming languages, embedded DSL, type-level programming, staging, Scala

1 Introduction

The Web is an appealing platform on which to write applications (*a.k.a.* Rich Internet Applications, RIA) because it makes them easy to deploy on clients and allows large scale innovative collaborative experiences [1,2]. RIAs are characterized by partial updates of the user interface (as opposed to refreshing the whole page with the classic hyperlink navigation), so a large part of the client-side code listens to user interface events (such as mouse clicks), triggers the appropriate action asynchronously on the server using AJAX [3] and updates the application's state and the page content with a new DOM fragment computed from the response data [1,4].

Writing large Web applications is known to be challenging [2,5], mainly because of the heterogeneous nature of the client-side and server-side environments [6,7]: writing distributed code requires its different parts to be consistent and leads to code duplication unless one uses the same language in both sides, which, in the case of the Web, means using JavaScript to write the whole application. However this language has several drawbacks making it hardly suitable for large code bases (*e.g.* no static typing, no module system, verbose syntax, *etc.*). Some other in-browser execution environments give the opportunity to write the client-side code in another language that JavaScript, *e.g.* Java applets [8], Adobe Flash [9] and Microsoft Silverlight[1]. However these technologies also have several drawbacks: they require an additional browser plugin to be

[1] http://www.microsoft.com/silverlight/

installed (which may not be available on all devices having a Web browser: for instance there is no way to execute Flash objects within an Apple smart phone), the page content cannot naturally be referenced by search engines, and the content is not structured in URLs that users can bookmark or share.

An increasing number of initiatives attempt to allow developers to write the client-side code in a language different than JavaScript but that can be compiled to JavaScript (*e.g.* GWT [10], Dart [11], TypeScript[2], Roy [12]). Some of these languages can also be compiled to another runtime environment usable on server-side (*e.g.* GWT, Dart, Kotlin[3], ClojureScript [13], Fay[4], Haxe [14] or Opa[5]). These languages are usually more suitable to write large applications either because they provide more constructs to build abstractions (such as object orientation), or because they are statically typed, or because they add some useful concepts that are missing in JavaScript (such as *adhoc* polymorphism or namespaces). Moreover, their ability to compile for both server and client sides usually allows developers to share some parts of code between server and client sides for free. However this shared code is restricted to exclusively use language constructs: concepts provided by external APIs can't be shared between server and client sides because the bindings with these environments are different on the server and client sides.

In this paper we investigate further *js-scala*, a path already introduced by Kossakowski *et al.* : defining JavaScript as a compiled embedded DSL in Scala [15]. This approach is based on Lightweight Modular Staging [16]: a Scala program written using the embedded DSL evaluates to an intermediate representation that can be further processed to perform domain specific optimizations and then to generate a JavaScript program (for the client-side) and a Scala program (for the server-side). This approach has two main advantages: (1) the embedding reduces the effort needed to define the language since we can reuse the Scala infrastructure and tooling, and (2) staging gives more knowledge to the compiler about how a given abstraction should be efficiently translated into the target environment. In other words any library-level abstraction can have the efficiency of a language-level abstraction.

This previous work on js-scala showed how to write client-side code in Scala using the JavaScript embedded DSL and how to share code between server-side and client-side. The authors also showed that the code was safer and more convenient to write thanks to Scala's static typing and class system. Finally, they showed how to *un-invert* the control of callback-based asynchronous APIs by using continuations but they did not address other concerns of RIAs development such as user interface events handling and DOM creation and manipulation.

This paper continues their work and presents several safe and expressive DSL constructs for RIAs development. We increase the expressivity of JavaScript native APIs with no runtime overhead and expose them through statically typed APIs. We make asynchronous programming simpler by raising asynchronous computations to first-class

[2] http://www.typescriptlang.org/
[3] http://kotlin.jetbrains.org/
[4] http://fay-lang.org/
[5] http://opalang.org/

citizens. We leverage the Scala type system to bring type-directed language features such as *ad hoc* polymorphism and type coercion to the client-side.

The next section introduces a simple application example showing the main challenges of RIA development, section 3 presents our DSL tackling these challenges, section 4 evaluates our solution, section 5 compares our solution to related works and section 6 concludes.

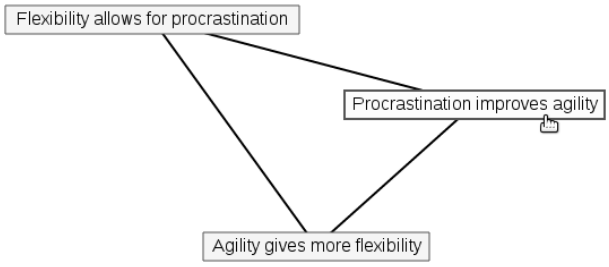
2 Background and Motivating Example

To illustrate the challenges of writing a RIA and the need for a DSL for writing them, we introduce a simple "mind mapping" application and show how the client-side code would typically be written in JavaScript.

2.1 Example Description

In our application, a mind map is represented as a graph where ideas or concepts are represented by vertices, and the relation between two ideas is represented by an edge. Users can visualize a mind map, zoom in and out and move it. They can also add ideas and link them together. Figure 1 shows the visualization of a simple map consisting of three ideas linked together, using our application.

Fig. 1. A simple mind map as visualized with our application



2.2 Implementation in JavaScript

Event handling RIAs distinguish themselves from classic Web pages by a higher degree of interactivity. In our example we want to let users zoom in and out a map using their mouse wheel. To implement this feature we need to attach an event handler for the `mousewheel` event on the DOM element containing the map. The following listing shows how to attach such an event handler using the native JavaScript API:

Listing 1. Native JavaScript API to handle events

```
mapElement.addEventListener("mousewheel", function (e) {
  scale = Math.round(scale + e.wheelDeltaY / 16);
  // ... update the user interface with the new scale
});
```

The event handler is simply a function taking the event data as a parameter. In the above example we use the `wheelDeltaY` property of the event to update the scale of the map.

We claim this code is fragile for two reasons. (1) The name of the event is passed as a string, so it is easy to mispell it. (2) The callback passed as a second parameter takes a parameter `e` whose fields vary according to the listened event, but developers have no way to check that the fields they're using are indeed defined on the event they're listening to. For instance, in our case we use the `wheelDeltaY` event field that is defined only on the `mousewheel` event.

Asynchronous programming Another characteristic of RIAs is that requests to the server are often performed asynchronously: instead of asking the browser to reload the whole page, the JavaScript code has to perform a request and to process the response, when available, to update a part of the user interface. The following listing shows how to send a request creating a vertex on the server and to insert it to the user interface:

```
var createVertex = function (text) {
  Ajax.post("/create", { content: text }, function (vertex) {
    addVertex(vertex);
  });
};
```

`Ajax.post` is a helper function that sends a HTTP request to the server and calls its last parameter (that is a *callback*) when the response data is available. The inverted control makes modularization harder to achieve: in the above listing the function creating the vertex is also responsible of updating the user interface. To relax this coupling the only option is to add a callback parameter to the `createVertex` function:

Listing 2. Callback-driven JavaScript APIs

```
var createVertex = function (text, callback) {
  Ajax.get("/create", { content: text }, function (data) {
    callback(data);
  });
};

createVertex("Hello, World!", function (vertex) {
  addVertex(vertex);
});
```

However increasing the number of callbacks makes the code flow harder to follow and adds distance to the programmer's initial intent.

Another issue with callback-driven programming arises when several dependent asynchronous computations are executed sequentially. Consider for example the code of listing 3 performing three consecutive Ajax requests. Notice that the code is getting deeper toward the right. This problem is often referred to as the "callback hell" [12]. This code is also hard to reason about: when will the second `console.log` statement be executed? Before or after `baz` has been fetched?

Listing 3. Sequential asynchronous calls

```
Ajax.get(fooUrl(), function (foo) {
  Ajax.get(barUrl(foo), function (bar) {
    Ajax.get(bazUrl(bar), function (baz) {
      console.log(foo + bar + baz);
    });
    console.log("bar = " + bar)
  });
});
```

DOM manipulation Updating the user interface usually means replacing a part of the DOM with another DOM fragment computed from data fetched by an AJAX request. This requires writing how to compute the new DOM fragment in JavaScript. For instance the following listing shows how to build a DOM tree representing a vertex in the mind map:

Listing 4. DOM fragment creation using the native API

```
var vertexDom = function (v) {
  var root = document.createElement("g");
  root.setAttribute("class", "vertex");
  root.setAttribute("transform",
    "translate(" + v.x + "," + v.y + ")");
  var rect = document.createElement("rect");
  rect.setAttribute("width", v.width);
  rect.setAttribute("height", v.height);
  var text = document.createElement("text");
  text.setAttribute("width", v.width);
  text.setAttribute("height", v.height);
  text.appendChild(document.createTextNode(v.content));
  root.appendChild(rect);
  root.appendChild(text);
  return root
};
```

For instance the following call:

```
vertexDom({
  x: 10, y: 10,
  width: 100, height: 60,
  content: "Hello, World!"
});
```

produces a DOM tree equivalent to the following HTML:

```
<g class=vertex transform="translate(10,10)">
    <rect width=100 height=60 />
    <text width=100 height=60>
        Hello, World!
    </text>
</g>
```

Not only the `vertexDom` function is very verbose, but it does not reflect the markup nested structure, making it hard to read and reason about.

Another way to build a DOM tree is to build a String containing the desired markup and then to ask the browser to parse it as HTML:

```
var vertexDom = function (v) {
  return '<g class=vertex ' +
              'transform="translate('+v.x+','+v.y+')">' +
            '<rect width='+v.width+' height='+v.height+' />' +
            '<text width='+v.width+' height='+v.height+'>' +
               v.content +
            '</text>' +
          '</g>'
};
```

The above code may be more readable, however this implementation is wrong: if we call it with a content containing brackets (*e.g.* `"<foo>"`), they won't be escaped and will produce a `foo` tag nested in the `text` tag, which is not the intended behavior. So, although this way reads slightly better it's not less error prone.

By the way, unlike the previous points, markup generation is a task that often needs to be performed from both server and client sides. From the server-side it produces HTML pages that search engines can crawl and from the client-side it produces DOM fragments that can be used to update the user interface. So we want to share HTML fragments definition between both server and client sides to get consistency in the rendering and to avoid duplication.

HTML template engines like Mustache[6] or Closure Templates[7] aim to make it simpler the definition of DOM fragments by providing a convenient syntax to describe the HTML structure and allowing the insertion of dynamic expressions in a safe way. Some template engines can be used on both client-side and server-side, however none is statically typed or has a practical expression language: dynamic content can only be provided as a map of key-value pairs (rather than typed values) and cannot be processed for presentational purposes from within the template.

2.3 Assessment

In the previous section we presented the three main tasks performed by the JavaScript code in RIAs: event handling, asynchronous programming and DOM manipulation. We noticed the following issues:

Lack of static checks Dynamic typing may give programmers more flexibility but can make it harder for little scripts to grow into mature and robust code bases and to perform refactorings. For instance, in the case of RIAs, a mispell in an event name may break the program behavior without anyone getting a sensible error message.

[6] http://mustache.github.com/

[7] https://developers.google.com/closure/templates/

Code hard to reason about and to modularize Callback-driven APIs are very common in JavaScript but their inverted control makes the code flow hard to follow and hampers modularization. Furthermore, native APIs are too verbose, making the code hard to read.

Sharing type safe code between clients and servers is hard to achieve The lack of static checks is even harder to tackle in the case of code shared between client and server sides. If we write the expression `xs.size == 0`, how should it be translated to the client-side environment (JavaScript) and to the server-side environment (*e.g.* the Java Virtual Machine)? Several solutions are valid, depending on the meaning of the expression: in JavaScript we could translate it as `xs.size === 0` or `xs.length === 0` if `xs` refers to a collection (in JavaScript the property to get the size of a collection is named `length`). So we need to define a proper expression language and to define how it should be translated for the client-side environment and the server-side environment, but such a task requires a high effort and is hard to achieve in an extensible way (so users can still integrate their own data types in expressions instead of being restricted to a set of supported types).

3 A DSL for Web Programming

JavaScript libraries could address some of the challenges presented above, but addressing type safety and asynchronous programming issues require language modifications (*e.g.* to bring a type system and a sequencing notation). Defining a language requires a high effort, especially to build development tools for the language so we chose to define our language as a compiled embedded DSL in Scala.

3.1 Introduction to Lightweight Modular Staging

Overview We use the Lightweight Modular Staging framework (LMS) to define our compiled embedded DSL. The main idea is that a program written using a DSL is evaluated in two stages (or steps): the first stage builds an intermediate representation of the program and the second stage transforms this intermediate representation into executable code (figure 2).

The bindings between stages are type-directed: a value of type `Rep[Int]` in the first stage will yield a value of type `Int` in the second stage. If you consider the following code:

```
val inc: Rep[Int] => Rep[Int] =
  x => x + 1
```

The function looks like a regular Scala function excepted that its parameter type and its return type are wrapped in the `Rep[T]` type constructor that denotes intermediate representations. The `+` operator has been defined on `Rep[Int]` values and returns the intermediate representation of an addition. Finally, the `inc` function returns the intermediate representation of a computation yielding the number following the value of the parameter `x`. You can get a `T` value from a `Rep[T]` value by generating code from the intermediate representation and compiling it:

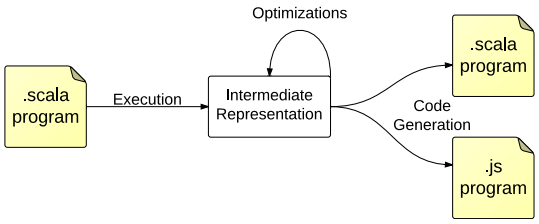


Fig. 2. Compilation of a program using LMS. An initial Scala program using embedded DSLs (on the left) evaluates to an intermediate representation from which the final program's code is generated (on the right).

```
val compiledInc: Int => Int =
  compile(inc)
```

The `compile` function takes a staged program of type `Rep[A] => Rep[B]` and returns a final program of type `A => B`.

The intermediate representation implementation is hidden for users but DSLs authors have to provide the corresponding intermediate representation of each construct of their language. For that purpose, LMS comes with an extensible intermediate representation implementation defining computations as a graph of statements. In the case of the `inc` function, this graph contains a `Plus` node applied on a `Sym` node (the `x` parameter) and a `Const` node (the literal value `1`).

Then, the code generation process consists in sorting this graph according to expressions dependencies and to emit the code corresponding to each node. The listings 5 and 6 show the JavaScript and Scala generated code for the `inc` function:

Listing 5. JavaScript code generation

```
var inc = function(x0) {
  var x1 = x0 + 1;
  return x1;
};
```

Listing 6. Scala code generation

```
def inc(x0: Int): Int = {
  val x1 = x0 + 1
  x1
}
```

Defining a DSL consists in three steps, each defining:

- The concrete syntax: an abstract API manipulating `Rep[_]` values ;
- The intermediate representation: an implementation of the concrete syntax in terms of statement nodes ;
- A code generator for the intermediate representation: a *pretty-printer* for each DSL statement node.

Example: null references 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 [17,18]. For example, consider the following typical JavaScript code finding a particular widget in the page and a then particular button in the widget:

Listing 7. 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 8. 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 8 but the expressiveness of listing 7. 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 4.1):

```
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 8 is generated.

The remaining of this section describes the design and the implementation of DSLs for Web programming using LMS.

3.2 Event Handling

We want to bring safety to the event handling API so that developers cannot mispell an event name and cannot attempt to read a property that is not defined on the type of the handled event. The difficulty comes from the fact that the type of the event passed to the handler varies with the event name. For instance, in listing 1 the handler processes `MouseWheelEvent` values because it listens to `mousewheel` events. Values of type `MouseWheelEvent` have a property `wheelDeltaY` but that's not the case of an event data of type `KeyboardEvent`, for example.

A possible solution could be to define a distinct method for each event instead of the single `addEventListener` method, so each method takes a handler with the according event type:

```
window.onKeyUp { e: Rep[KeyboardEvent] =>
  println(e.key)
}
window.onMouseWheel { e: Rep[MouseWheelEvent] =>
  println(e.wheelDeltaY)
}
```

In the above listing, the `onKeyUp` method attaches an event handler for the `keyup` event that uses values of type `KeyboardEvent`, and the `onMouseWheel` method does the same for `mousewheel` events that use values of type `MouseWheelEvent`.

However, implementing this solution requires a high effort because we have to define as many methods as there are events. We want to provide a single polymorphic method similar to the native API. To achieve that, we need to encode the dependency relation between an event name and its corresponding event type at the type-level. This dependency can naturally and elegantly be encoded using dependent method types [19]:

```
class EventDef {
  type Data
}

def on(ev: EventDef)(handler: Rep[ev.Data] => Rep[Unit]): Rep[Unit]
```

(We renamed `addEventListener` to `on`, for the sake of brevity). An `EventDef` value represents an event that carries its corresponding event data type in its `Data` type member. The `on` method takes a parameter `ev` of type `EventDef` and a `handler` parameter whose type refers to the `Data` member of the `ev` parameter, so the type of the handler depends on the `ev` value. We can then define the `keyup` and `mousewheel` events as follows:

```
object KeyUp extends EventDef { type Data = KeyboardEvent }
object MouseWheel extends EventDef { type Data = MouseWheelEvent }
```

The following Scala listing is equivalent to the JavaScript listing 1 and is completely type safe: if the user misspells the event name or tries to use an undefined property on an event his program won't compile.

```
window.on(MouseWheel) { e =>
  scale = Math.round(scale + e.wheelDeltaY / 16)
}
```

3.3 Asynchronous Programming

The previous work on js-scala [15] 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 parts.

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 9 shows how the listing 2 can be re-written using our DSL. The `createVertex` function now returns an asynchronous value instead of taking a callback as parameter. Then, the **`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 9. Asynchronous values are first class citizen

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

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

Listing 10. 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)
```

The listing 10 translates the callback hell example (listing 3) 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 will be executed, and so on. There is no nested callbacks and the order of execution is directly reflected by the order of statements.

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 transformed using the `map` and `flatMap` methods, turning them into first-class citizens: functions can take them as parameters and return them.

Listing 11. Parallel computations in Scala

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

Listing 12. Generated JavaScript code

```
var x1 = new Promise();
AjaxGet("/foo", x1);
var x2 = new Promise();
AjaxGet("/bar", x2);
x1.onComplete(function (foo) {
 x2.onComplete(function (bar) {
  var x3 = foo + bar;
  console.log(x3);
 });
});
```

Listing 11 defines two asynchronous computations running in parallel and listing 12 shows the generated JavaScript code (we renamed some identifiers for the sake of readability). The generated `Promise` constructor code has been omitted, it creates an object holding a list of callbacks to call when the asynchronous value is completed. The `AjaxGet` function code has also been omitted, it creates a `XMLHttpRequest` object that sends an HTTP request and completes its promise parameter with the response, when received.

3.4 DOM Definition

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 13 uses our DSL and generates a code equivalent to listing 4. 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.

Listing 13. 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)
        )
    )
```

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 13 produces the following Scala code usable on server-side (the generated code for client-side is roughly equivalent to listing 4):

```
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 2.3 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.

3.5 Leveraging the Scala Type System on the Client-Side

This section presents some more general purpose improvements of JavaScript based on type-directed mechanisms of Scala.

***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 [21], but this solution is painful for developers and incurs a performance overhead.

We propose to achieve *ad hoc* polymorphism using typeclasses [22,23,19] 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 14 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.

Type coercion JavaScript comes with only one numeric type, `Number`, that represents double-precision 64-bit values. If developers want to use another numeric data type, *e.g.* representing rational numbers, they can't use the native arithmetic operators (`+`, `-`, *etc.*) on them, making the code less readable because the same concepts (*e.g.* adding two numeric values) are described by different operators or methods. Furthermore, the code is also harder to write when it comes to mix different types of numeric values. Indeed, this requires to manually take care of their compatibility: before adding a rational number with a `Number` value, developers have to convert the former in the latter's type. Consider as an example the listing 15 mixing rational numbers and native numbers.

Listing 15. Mixing integer and floating-point arithmetic in JavaScript.

```
// i and j are rational numbers, x and y are native numbers
var main = function (i, j, x, y) {
  console.log(rat_plus(i, j));
  var m = rat_toNumber(i) + y;
  var n = x + rat_toNumber(j);
  console.log(m + n);
};
```

Our DSL solves these problems by defining numeric operations in a safe and extensible way: each operation is a polymorphic method and operand coercion is expressed using functional dependencies [24]. It allows developers to use a homogeneous syntax for arithmetic operations with different numeric types and to mix heterogeneous types in a same operation. For example, the add operation is defined as follows:

```
def numeric_plus[A : Numeric](lhs: Rep[A], rhs: Rep[A]): Rep[A]
```

The `A : Numeric` syntax means that the function is polymorphic on any type `A`, as long as there is an available `Numeric[A]` value.

Then we add a `+` method to any `Rep[_]` value, using an implicit class, so we can use an homogeneous syntax to write arithmetic operations whatever the numeric type of values this operation is applied on:

```
implicit class NumericOps[A : Numeric](lhs: Rep[A]) {
  def + (rhs: Rep[A]) = numeric_plus(lhs, rhs)
}
```

Listing 14. 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: Double }
// 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[Article]]) = {
  document.body.append(listWidget(users))
  document.body.append(listWidget(articles))
}
```

However, the above code does not handle operands coercion yet. To do so, we want to express that an expression `a + b` is valid either if `a` and `b` have the same numeric type, or if one of them has a numeric type and the other can be converted to this numeric type. The result type of such an expression depends on the types of the operands, *e.g.* an integer and a double value produce a double value, two integer values produce another integer value. We express these constraints using functional dependencies:

```
implicit class NumericOps(lhs: Rep[A]) {
  def + [B, C](rhs: Rep[B])(implicit ev: (A ~ B) ~> C): Rep[C] =
    numeric_plus(ev.lhs(lhs), ev.rhs(rhs))
}
```

The `ev` parameter has type `(A ~ B) ~> C`, which means *combining a `A` and a `B` gives a `C`*. Then we need to define which combinations of `A`, `B` and `C` are valid:

Listing 16. Numeric operands constraints

```
implicit def sameType[A : Numeric] =
  new ((A ~ A) ~> A)(identity, identity)
implicit def promoteLhs[A, B : Numeric](implicit aToB: A => B) =
  new ((A ~ B) ~> B)(a => convert[B](a), identity)
implicit def promoteRhs[A, B : Numeric](implicit aToB: A => B) =
  new ((B ~ A) ~> B)(identity, a => convert[B](a))
```

The `sameType` value says that two values of type `A` give another value of type `A`. The `promoteLhs` value says that a value of type `A` and a value of a numeric type `B` can be used as operands as long as there is a way to convert the value of type `A` to a value of type `B` (so the left hand side is promoted to the type `B`). The `promoteRhs` does the same for the right hand side.

The listing 15 can be rewritten as shown in listing 17 with our DSL. The `+` operator can be safely used to add rational and floating-point numbers and type promotion is automatically handled by the type system. It is worth noting that the `Rational` data type can be written independently of the type coercion system.

Listing 17. Type-coercion automatically handled by our DSL

```
def main(i: Rep[Rational], j: Rep[Rational],
         x: Rep[Double], y: Rep[Double]) = {
  println(i + j)
  // i is promoted to Rep[Double], m has type Rep[Double]
  val m = i + y
  // j is promoted to Rep[Double], n has type Rep[Double]
  val n = x + j
  println(m + n)
}
```

3.6 Discussion

Benefits of the compiled embedded approach We implemented our language as a compiled embedded DSL in Scala. The process of generating code from a DSL is usually described as a two steps process: a program first evaluates to an intermediate representation and then the final program's code is generated from this intermediate representation, with the opportunity to generate efficient code by applying domain specific

optimizations. We think it's worth considering a third step, which occurs before the program evaluation: its compilation. This step affects DSLs design because their embedded nature let them reuse the host language features (*e.g.* the type system or syntactic sugars) to implement their own features. The following paragraph gives an overview of the different mechanisms used to implement our DSLs.

First, features like type coercion and *ad hoc* polymorphism only rely on the type system, so they operate during the compilation. Then, during the evaluation, the DSL constructs build an intermediate representation of the program, for instance the DOM definition DSL builds a tree of DOM nodes. Last, the code generator produces the final program using the domain specific information available in the intermediate representation to generate efficient code, for example the Option monad code generator produces code checking against nullity before dereferencing a value.

An important consequence of the implementation as compiled embedded DSLs is the low effort required to be able to share code between server and client sides. The other ways to define DSLs compiling to multiple backends are either to define a brand new language so the compilation generates appropriate code for each target (*e.g.* Opa), or to define a library within a host language that already compiles to several targets (*e.g.* GWT, Kotlin). However, the first way requires a high effort or leads to defining several small languages that are hard to integrate seamlessly, and the second way does not give the opportunity to specialize the generated code according to the targets (so you can't generate, from the same library, code building DOM fragments on the client-side and code building XML trees on the server-side).

Compiled embedded DSLs have the advantages of both approaches because they are simply defined as libraries but they let developers specialize the generated code according to each target. This opportunity made it possible to define, as a library, the HTML templating DSL generating code building DOM fragments on client-side and code building XML trees on server-side.

Lack of a data type definition DSL On the other side of the coin, because that is not possible to define a library without defining how to translate it on both server and client sides, there is no simple way to let DSLs users define their own libraries. This feature comes almost for free with approaches like GWT, Fay or Kotlin because they derive the client-side program and the server-side program from the initial program source (that contains data type definitions such as classes definitions). In our case, the client-side program and the server-side program are derived from the intermediate representation produced by the evaluation of the initial program. So, only concepts reified in an intermediate representation during this evaluation can be part of the final programs. However regular Scala's classes definitions are not reified in an intermediate representation (it would mean that the constructor of a type `A` would yield a value of type `Rep[A]`). Scala-virtualized [25] provides such a mechanism but is limited to record types and provides only projection functions, so for more complex cases users have to write some repetitive and boilerplate code.

In summary, the compiled embedded approach may require more work to define custom data types but gives control on the way these data types will be encoded for

each target platform (*e.g.* the JavaScript code generator uses native arrays to encode the lists of the list manipulation DSL).

However, the next release of Scala will have support for macros making it possible to generate reified data types from a concise syntax without even relying on a compiler extension such as Scala-virtualized.

Custom code generation to handle browsers incompatibilities Each component of a DSL is defined in a trait and each corresponding code generator for a given target is also defined in a trait. The Scala language allows developers to mix several traits together, so they can build their language by picking the DSL traits they want to use.

Furthermore, a code generator can be tweaked by defining a sub-trait specializing some of its methods. This feature can be used to handle cross-browser incompatibilities. For instance we have written code generators specializing the output for some statement nodes in order to handle Internet Explorer incompatibilities.

4 Implementation and Evaluation

4.1 Implementation

This section presents relevant implementation details of the js-scala DSL[8].

Type-directed DSL constructs The *ad hoc* polymorphism feature of our DSL is directly borrowed from Scala's typeclasses encoding but the type coercion feature required to define a type-level DSL to express constraints. Numeric operations take an implicit value of type `(A ~ B) ~> C`. This type is actually a syntactic sugar for the type `~>[~[A, B], C]` (Scala supports an infix notation for type constructors of arity two). The `~>` type also takes two value parameters, saying how to obtain a value of the target type for each operand. Here is the definition of the `~>` type:

```
class ~>[A <: Args, B : Numeric](
  val lhs: Rep[A#Lhs] => Rep[B],
  val rhs: Rep[A#Rhs] => Rep[B]
)
trait Args { type Lhs; type Rhs }
type ~[A, B] = Args { type Lhs = A; type Rhs = B }
```

The `~` type is just an alias to the `Args` type to get the infix notation. The `Args` type aggregates the types of the operands (`Lhs` and `Rhs`) involved in a binary operation.

When a numeric operation is used, the Scala compiler fills the required implicit parameter of type `(A ~ B) ~> C` by looking for an implicit value of this type that satisfies the operands types and fixes the `C` type parameter value. The defined implicit values are given in lisiting 16. For example, for a numeric operation involving a `Rep[Int]` value and a `Rep[Double]` value, the `sameType` value is not applicable because its `A` type parameter cannot match both `Int` and `Double`, the `promoteRhs` value is not applicable because there is no implicit conversion from `Double` to `Int` but the `promoteLhs`

[8] The complete implementation is available at http://github.com/js-scala

value is applicable and fixes the `C` type parameter to `Double`. So, such an operation would promote the `Rep[Int]` parameter into a `Rep[Double]` value and would return a `Rep[Double]` value.

Intermediate representation construction Every DSL construct producing code in the final program is first reified in an intermediate representation. Listing 18 shows extracts of both the abstract DSL interface of the **`null`** reference handling DSL and its implementation building an intermediate representation of values dereferencing.

Listing 18. Null reference handling DSL

```
// DSL interface
trait OptionOps { this: Base =>
  implicit def repOptionToOps[A](o: Rep[Option[A]]) =
    new OptionOpsCls(o)
  class OptionOpsCls[+A](o: Rep[Option[A]]) {
    def isEmpty = option_isEmpty(o)
    def foreach(f: Rep[A] => Rep[Unit]) =
      option_foreach(o, f)
  }

  def option_isEmpty[A](o: Rep[Option[A]]): Rep[Boolean]
  def option_foreach[A](o: Rep[Option[A]],
      f: Rep[A] => Rep[Unit]): Rep[Unit]
}
// Intermediate representation
trait OptionOpsExp extends OptionOps { this: EffectExp =>
  def option_isEmpty[A](o: Exp[Option[A]]) = OptionIsEmpty(o)
  def option_foreach[A](o: Exp[Option[A]],
      f: Exp[A] => Exp[Unit]) = {
    val a = fresh[A]
    val b = reifyEffects(f(a))
    reflectEffect(OptionForeach(o, a, b), summarizeEffects(b).star)
  }

  case class OptionIsEmpty[A](o: Exp[Option[A]]) extends Def[Boolean]
  case class OptionForeach[A](o: Exp[Option[A]],
      a: Sym[A], block: Block[Unit]) extends Def[Unit]
}
```

We limited the DSL to two operations (`foreach` and `isEmpty`) for the sake of brevity. The `OptionOps` trait defines the user facing DSL, it enriches `Rep[Option[_]]` values by providing an implicit conversion to `OptionOpsCls` that defines the `isEmpty` and `foreach` methods that respectively tests the emptiness of a value and derefences a value if not empty. These methods provide a convenient syntax to use `Rep[Option[_]]` values and delegate to the `option_isEmpty` and `option_foreach` operations, re-

spectively, that are left abstract. The `OptionOpsExp` trait implements the `option_isEmpty` and `option_foreach` methods. The `option_isEmpty` implementation simply returns a statement node of type `OptionIsEmpty`, but the `option_foreach` implementation is more subtle because it has to track the dependencies of its function parameter `f` (for instance, side-effects performed within the `f` function must be reflected by the returned `OptionForeach` node).

The principle is the same for all DSLs: first, provide an abstract trait with the DSL vocabulary, then implement each DSL construct to return a reified intermediate representation of itself. Sometimes you have the opportunity to perform some optimizations at the evaluation level such as constants folding. For instance, the DOM definition DSL builds trees consisting of a root node and eventually several children. If these children are known at evaluation time we can build a node having a `List[Rep[Node]]` value as children instead of a `Rep[List[Node]]` so the generated code will be able to inline the loop adding the children to the parent node. Listing 19 shows the relevant part of code checking if the `children` parameter of the `forest_tag` method is a constant list (the `ListNew` statement node represents constant lists) and building different intermediate representations according to this test.

Listing 19. Optimized intermediate representation for the DOM definition DSL

```
trait ForestExp extends Forest with EffectExp { this: ListOpsExp =>
  def forest_tag(name: String, attrs: Map[String, Exp[String]],
      children: Exp[List[Node]]) = children match {
    case Def(ListNew(children)) =>
      Tag(name, Left(children.toList), attrs)
    case _ =>
      Tag(name, Right(children), attrs)
  }
  case class Tag(name: String,
      children: Either[List[Exp[Node]], Exp[List[Node]]],
      attrs: Map[String, Exp[String]]) extends Def[Node]
}
```

Code generation 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 20 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`**.

Listing 20. 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)
    }
}
```

4.2 Evaluation

Our goal was to define, with a minimal effort, a language for RIAs development. The main challange was to provide programming language constructs both safer and more expressive without abstraction penalty. We focused on the main concerns of RIA development: event handling, asynchronous programming and DOM fragments definition, as well as on more general programming concerns: `null` reference handling and type safety.

Using our DSL, we were able to completely rewrite the client-side part of our mind mapping application and to integrate it in a mainstream Web framework for the server-side part. The number of lines of code needed to write the DOM fragments reduced by 38%. The size of code parts dealing with events did not change but gained type safety. Finally, code handling asynchronous computations were easier to modularize.

An interesting side-effect of the design of our DSL as a compiled embedded DSL is the ability to reuse some code between both client and server sides. We were able to reuse DOM fragments definition on both sides and to build two variants of the application with minimal effort. The first variant renders HTML from the server-side and the second variant renders DOM fragments from the client-side. The flexibility to decide later where to run the logic (on client-side or on server-side) gives more agility in the development because we can easily adapt the application. For instance we can choose to perform the rendering on server-side if the application targets smartphones with a small processing speed or to perform the rendering on client-side, to preserve the server CPU usage, if the application targets desktop computers.

All these benefits came with no runtime overhead, excepted for the asynchronous programming DSL that creates, for each asynchronous computation, a `Promise` value registering the callbacks to call when the value is available.

5 Related Works

The most popular JavaScript library, jQuery [26], used by more than 40% of the top million sites[9] aims to simplify RIA development. It handles the `null` references problem by wrapping each query result in a container so before each further method call it tests the emptiness of the container and applies effectively the operation only if the container is not empty. It provides an expressive API but the emptiness checking involves a slight performance penalty since each result is wrapped and then unwrapped for the subsequent method invocation. jQuery also has an API to turn asynchronous computations into first-class citizens. However, jQuery is just a library and thus cannot bring type safety to JavaScript programs.

TypeScript[10] is a language compiling to JavaScript supporting classes, modules and soft typing. Because TypeScript is a superset of JavaScript, any valid JavaScript program is a valid TypeScript program. Developers can progressively leverage TypeScript's features to turn their JavaScript sources into large, modular and more robust code bases. However, TypeScript does not address problems specific to RIAs development like DOM fragments definition or asynchronous programming.

The Roy [12] programming language is a statically typed functional programming language compiling to JavaScript. It features global type inference, algebraic data types, pattern matching and a sequencing notation similar to Scala's `for` notation. Roy does not address concerns such as event handling and DOM fragments definition.

GWT [10] compiles Java code to JavaScript. It exposes the browser's APIs through type safe Java APIs and simplifies DOM fragments definition. However, it still uses a callback-driven programming model for asynchronous computations.

Kotlin[11] is a recent language written by JetBrains, that targets both the JVM and JavaScript. It has libraries addressing partially Web programming concerns and supports advanced programming language features such as mixins, type inference, variance annotations for type parameters and pattern matching.

Opa[12] is a language and a full stack designed for Web programming. It features static typing with global type inference, modules and pattern matching. The language provides a syntactic sugar for writing HTML fragments in the code. The server and client parts of an application can be written in the Opa language, the system can decide whether a code fragment will run on server-side, client-side or both. However, the Opa language is not extensible, so there is no way for developers to customize the compilation process of their abstractions on client and server sides.

Dart [11] is a language designed specifically for Web programming. It has been designed to have a familiar syntax for JavaScript developers and supports soft typing, classes and a module system. It also provides a high-level concurrent programming system similar to an actor system. Finally, Dart code can be executed on both client and server sides but the compiler is not extensible so developers cannot customize the compilation process of a given abstraction on client and server sides.

[9] http://trends.builtwith.com/javascript
[10] http://www.typescriptlang.org/
[11] http://kotlin.jetbrains.org/
[12] http://opalang.org/

Table 1 gives a synthetic view of the comparison of our work with other approaches. Lines give features desired to write large Web applications and colums list mainstream tools for Web programming. Cells show the support level of each feature for each tool.

	jQuery	TypeScript	Roy	GWT	Kotlin	Opa	Dart	js-scala
Type safe event handling				***				***
DOM definition	*			***	**	***	*	***
Asynchronous programming	**		***			***	**	***
Null references	**		***		*	**		***
Type coercion				*	*	*	*	***
Static typing		*	**	**	**	**	*	**
Easiness for defining libraries	**	***	***	***	***	***	***	*
Client-server code sharing		*	*	*	*	*	*	***

Table 1. Support level of Web programming features by mainstream tools. Zero star means that the feature is not supported at all, three stars mean a strong support.

6 Conclusion and Perspectives

Writing RIAs is challenging because they require a lot of logic on the client-side but the JavaScript language and the Web browser's native APIs make the code fragile and hard to reason about and to maintain.

We improved the js-scala DSL by adding first-class support for RIA concerns such as events handling, DOM fragments definition and asynchronous programming. Our work provides a safer and more expressive language than JavaScript and its native APIs to write RIAs but has no runtime overhead.

Future work will focus on the ability to choose where to execute a piece of code (on server-side or on client-side) in order to be able to delay this choice until the runtime.

References

1. J. Farrell and G. Nezlek, "Rich internet applications the next stage of application development," in *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, pp. 413–418, june 2007.
2. 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*, (Washington, DC, USA), pp. 319–328, IEEE Computer Society, 2008.
3. J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.
4. N. K. Marianne Busch, "Rich internet applications. state-of-the-art.," Tech. Rep. 0902, Ludwig-Maximilians-Universität München, 2009.
5. J. C. Preciado, M. L. Trigueros, F. Sánchez-Figueroa, and S. Comai, "Necessity of methodologies to model rich internet applications," in *WSE*, pp. 7–13, IEEE Computer Society, 2005.

6. R. Rodríguez-Echeverría, "Ria: more than a nice face," in *Proceedings of the Doctolral Consortium of the International Conference on Web Engineering*, vol. 484, CEUR-WS.org, 2009.
7. 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, (New York, NY, USA), pp. 647–652, ACM, 2009.
8. E. Anuff, *The Java sourcebook: a complete guide to creating Java applets for the Web*. John Wiley & Sons, Inc., 1996.
9. H. Curtis, *Flash Web Design: the art of motion graphics*. New Riders Publishing, 2000.
10. P. Chaganti, *Google Web Toolkit: GWT Java Ajax Programming*. Packt Pub Limited, 2007.
11. R. Griffith, "The dart programming language for non-programmers-overview," 2011.
12. B. McKenna, "Roy: A statically typed, functional language for javascript," *Internet Computing, IEEE*, vol. 16, pp. 86 –91, may-june 2012.
13. M. McGranaghan, "Clojurescript: Functional programming for javascript platforms," *Internet Computing, IEEE*, vol. 15, no. 6, pp. 97–102, 2011.
14. N. Cannasse, "Using haxe," *The Essential Guide to Open Source Flash Development*, pp. 227–244, 2008.
15. G. Kossakowski, N. Amin, T. Rompf, and M. Odersky, "JavaScript as an Embedded DSL," in *ECOOP 2012 – Object-Oriented Programming* (J. Noble, ed.), vol. 7313 of *Lecture Notes in Computer Science*, (Berlin, Heidelberg), pp. 409–434, Springer Berlin Heidelberg, 2012.
16. T. Rompf, *Lightweight Modular Staging and Embedded Compilers*. PhD thesis, IC, Lausanne, 2012.
17. T. Hoare, "Null references: The billion dollar mistake," *Presentation at QCon London*, 2009.
18. M. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 133–143, IEEE, 2009.
19. B. C. Oliveira, A. Moors, and M. Odersky, "Type classes as objects and implicits," *SIGPLAN Not.*, vol. 45, pp. 341–360, Oct. 2010.
20. N. Zakas, *Maintainable JavaScript*. O'Reilly Media, 2012.
21. H. Abelson and G. Sussman, "Structure and interpretation of computer programs," 1983.
22. 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, (New York, NY, USA), pp. 60–76, ACM, 1989.
23. M. Odersky, "Poor man's Type Classes," in *Presentation at the meeting of IFIP WG*, July 2006.
24. M. Jones, "Type classes with functional dependencies," in *Programming Languages and Systems* (G. Smolka, ed.), vol. 1782 of *Lecture Notes in Computer Science*, pp. 230–244, Springer Berlin Heidelberg, 2000.
25. A. Moors, T. Rompf, P. Haller, and M. Odersky, "Scala-virtualized," in *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pp. 117–120, ACM, 2012.
26. B. Bibeault and Y. Kats, *jQuery in Action*. Dreamtech Press, 2008.