Statically typed Web programming

Julien RICHARD-FOY, Olivier BARAIS, and Jean-Marc JÉZÉQUEL
{first}.{last}@irisa.fr

INRIA

Abstract. Rich Internet Applications involve 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 several inadequacies (no static typing, no module system, verbose syntax, *etc.*). 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 a 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 DOM 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 parts to be consistent together and leads to duplication unless you use 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 inadequacies making it hardly suitable for large code base (*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 than JavaScript, *e.g.* Java applets [8], Adobe Flash [9] and Microsoft Silverlight[1]. However these technologies have several drawbacks: they require an additional browser plugin to be installed (which may not be available on all devices having a Web browser: for instance there is

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

no way to execute Flash objects within an Apple smart phone), the page content can't 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 of 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 a bunch of 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 use exclusively 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 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.

The previous work 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, DOM creation and manipulation, and application's state management.

This paper continues their work and presents safe and expressive DSL constructs for RIAs development.

The next section introduces a simple application example showing challenges of RIA development, section 3 presents our DSL tackling these challenges, section 4 validates our solution and compares it to related work and section 5 concludes.

---

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

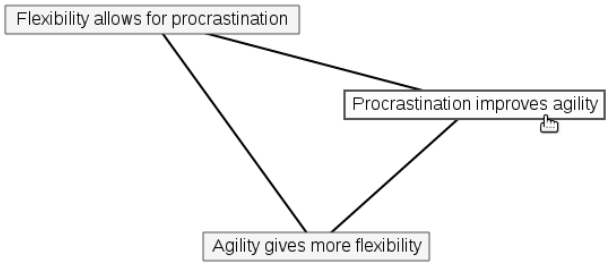
2 Background and motivating example

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

Events handling RIAs distinguish 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 1.1. 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 change the scale of the map.

We think 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 to updating the user interface. If we want to relax this coupling the only option is to add a callback parameter to the `createVertex` function:

Listing 1.2. 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 following listing performing three consecutive Ajax requests:

Listing 1.3. 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)
  });
});
```

Notice that the code is getting deeper toward the right, this problem is often referred to as the "callback-hell". This code is also hard to reason about: when will the second `console.log` statement be executed? Before or after `baz` has been fetched?

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 to write 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 1.4. 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 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 has a practical expression language (dynamic content can only be provided as a map of key-value pairs and no operation can be done on values) or is statically typed.

2.3 Assessment

In the previous section we showed the three main tasks performed by the JavaScript code in RIAs: events 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 getting a sensible error message.

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 bothers modularization. Furthermore, native APIs are too much 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` in a template, how should it

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

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

be translated in JavaScript and in Scala? 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`). We need to define a proper expression language and to define how it should be translated into JavaScript and Scala, but such a task requires a high effort and is hard to achieve in an extensible way (so users can still use their own data types in expressions instead of being restricted to a set of supported types).

3 DSL design and implementation

This section presents both the design and the implementation of a compiled embedded DSL addressing the problems presented in the previous section.

3.1 Introduction to Lightweight Modular Staging

We chose to implement our DSL as a compiled embedded DSL in Scala using Lightweight Modular Staging. The embedding lets us reuse the host language infrastructure (compiler, development tools) and type system, and staging gives the opportunity to generate efficient code from expressive code by applying domain specific optimizations.

Lightweight Modular Staging (LMS) is a Scala framework for defining compiled embedded DSLs. 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).

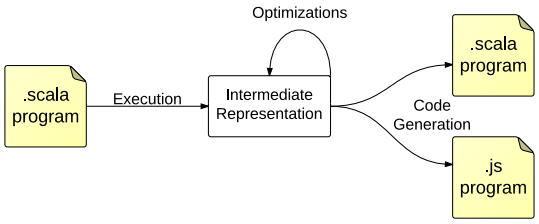


Fig. 2. Compilation of a program using LMS

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 `inc` function returns the intermediate representation of a computation yielding the number following the value of the parameter `x` (that itself is the intermediate representation of a number). 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. You can get a `T` value from a `Rep[T]` value by generating code from the intermediate representation and compiling it:

```
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 construction of their language. For that purpose, LMS comes with an extensible intermediate representation implementation defining computations as a graph of statements. 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 1.5 and 1.6 show the JavaScript and Scala generated code for the `inc` function:

Listing 1.5. JavaScript code generation

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

Listing 1.6. 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.

3.2 Events handling

We want to bring safety to the events handling API so that developers can't mispell an event name and can't 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. A value of type `KeyboardEvent` contains a `key` field returning the value of the key involved in the event, and a value of type `MouseWheelEvent` contains a `wheelDeltaY` field returning the value of the mouse button involved in the event. For instance in the listing 1.1, the handler processses `MouseWheelEvent` values because it listens to `mousewheel` events.

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)
}
```

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 at the type-level the dependency relation between an event name and its corresponding event type. This dependency can naturally and elegantly be encoded using dependent method types [17]:

```
class EventDef {
  type Type
}

def on(ev: EventDef)(handler: Rep[ev.Type] => 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 type in its `Type` type member. The `on` method takes a parameter `ev` of type `EventDef` and a `handler` parameter whose type refers to the `Type` 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 Type = KeyboardEvent }
object MouseWheel extends EventDef { type Type = MouseWheelEvent }
```

The following Scala listing is equivalent to the JavaScript listing 1.1 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

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

An asynchronous value of type `Rep[A]` is modelled by a value of type `Rep[Future[A]]`. This type has the following methods:

- `foreach(f: Rep[A] => Rep[Unit]): Rep[Unit]`, registering a callback to eventually do something with the value when available ;
- `map(f: Rep[A] => Rep[B]): Rep[Future[B]]`, to eventually transform the value when available ;
- `flatMap(f: Rep[A] => Rep[Future[B]]): Rep[Future[B]]` to eventually transform the value when available ;

Asynchronous values can be transformed using the `map` and `flatMap` methods, turning them into first-class citizen: functions can take as parameters and return asynchronous values.

The listing 1.2 can be re-written as follows with our DSL:

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

// Usage
val vertexAsync = createVertex("Hello, World!")
for (vertex <- vertexAsync) {
  addVertex(vertex)
}
```

The `createVertex` now returns an asynchronous value instead of taking a callback as parameter.

The "callback-hell" example (listing 1.3) can be re-written as follows:

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

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.

The following listing shows the generated JavaScript code for an example where two asynchronous computations run in parallel (we renamed some identifiers for the sake of readability):

Listing 1.7. Parallel computations in Scala

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

Listing 1.8. 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);
 });
});
```

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, as an embedded DSL, with a small effort, a template engine that is statically typed, able to insert dynamic content in a safe way,

that provides a powerful expression language, requires no extra compilation step and that 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. The following listing uses our DSL and generates a code equivalent to the listing 1.4:

Listing 1.9. 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)
        )
    )
```

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.

Reuse the DOM definition DSL from server-side Our DSL is equivalent as a template engine with Scala as the expression language. Making it usable on both server and client sides is as simple as defining another code generator for the DSL, producing Scala code. The result is a lightweight statically typed template engine, seamlessly integrated with the other code parts (because it is defined as an embedded DSL) and with a convenient expression language. For instance, the template written in the listing 1.9 produces the following Scala code usable on server-side (the generated code for client-side is roughly equivalent to the listing 1.4):

```
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
}
```

3.5 More general purpose improvements of JavaScript

This section presents some more general purpose improvements of JavaScript.

Null references Null references are a known source of problems in programming languages [18,19]. For example, consider the following typical code finding a particular widget in the page and a then particular button in the widget:

Listing 1.10. 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 1.11. 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) { ... });
  }
}
```

Our solution has both the safety and performance of the listing 1.11 and the expressiveness of the listing 1.10. We encode the potential emptiness of a value of type `Rep[A]` using the `Rep[Option[A]]` type at the stage level but we don't generate code wrapping values in a container, instead we just check if a value is **`null`** or not. Here is a listing that uses our DSL and compiles to code equivalent to the listing 1.11:

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

State monad Managing the state of an application is cumbersome and fragile in JavaScript due to the lack of encapsulation constructs. For example, consider the following function incrementing a counter and returning its previous value:

Listing 1.12. Fragile state handling

```
var x;
var inc = function () {
```

```
  var prev = x;
  x = x + 1;
  return prev
};

// Usage
x = 42;
inc();
var y = inc();
alert(y);
```

This code suffers from two problems: the variable `x` is global and if the user calls the `inc` function without initializing first the `x` variable the behavior will be unexpected.

To workaround these issues, developers can wrap the variable declaration and the `inc` function code in a factory function. Thus the state is hidden and developers may not forget the initialization:

Listing 1.13. State encapsulation within a function

```
var incFactory = function (init) {
  var x = init;
  return function () {
    var prev = x;
    x = x + 1;
    return prev
  }
};

// Usage
var inc = incFactory(42);
inc();
var x = inc();
alert(x);
```

The code is now safer, but it is also bigger, and it still uses a mutable state. It's known that mutable objects in the code make this one harder to reason about [20,21]. Getting rid of mutable objects is simple: instead of modifying a state, a function takes its initial value as a parameter and returns its new value in its result:

Listing 1.14. Explicit state threading

```
var inc = function (state) {
  return {
    state: state + 1,
    value: state
  }
};

// Usage
var x1 = inc(42);
var x2 = inc(x1.state);
alert(x2.value);
```

However, explicitly threading the state in all functions parameters is not desired because of the syntactic and performance cost of adding an extra parameter to each function and of returning two results instead of one. Furthermore, manually passing the result of each previous call to the next one is error prone.

The State monad [22] is a functional programming pattern that avoids to explicitly thread the state in functions using it or modifying it. In Scala, the **`for`** notation provides an elegant way to sequence monadic computations, and thus to implicitly thread a state across computations, as shown in the following listing:

Listing 1.15. Implicitly threaded state using the State Monad

```
val inc = for {
  x <- get[Int]
  _ <- put(x + 1)
} yield x

// Usage
val usage = for {
  _ <- inc
  x <- inc
} yield alert(x)
usage.init(42)
```

The `get` and `put` primitives are part of the State monad DSL, they allow to retrieve the current state and to change it. The `usage` value is a reusable piece of program that calls two times the `inc` function and prints the result of the second call. The last line of the listing runs the `usage` program on an initial state.

The State monad programming model has the advantages but not the shortcomings of both listings 1.13 and 1.14 because there is no global variable (the state is implicitly threaded thanks to the State monad but state modifications explicitly use the `get` and `put` primitives) and everything is immutable.

Furthermore, our DSL implementation produces an intermediate representation semantically equivalent to the monadic sequences but using regular variables, so that there is no abstraction penalty: the listing 1.15 compiles to code equivalent to the listing 1.12.

(TODO More details about this optimization)

***Adhoc* polymorphism** Because of the dynamically typed nature of JavaScript, when calling a function there is no proper way to call 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 *adhoc* polymorphism on JavaScript objects consists in defining the polymorphic function on the prototypes of the objects. However, modifying existing object prototypes is considered to be a bad practice [23]. Another way could consist in manually code the dispatch logic, by registering supported data types at the beginning of the program execution, as described in section 2.4.3 of [24], but this solution is painful for developers and incurs a performance overhead.

We propose another way to achieve *adhoc* polymorphism that is entirely type-directed (the dispatch happens at compile-time rather than at runtime), with no impact on the generated code and supporting retroactive extension, using typeclasses [25,26,17].

The listing 1.16 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`.

Type coercion (TODO More general problem description)

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 integer values or arbitrary length values, 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 an arbitrary length value with a `Number` value, developers have to convert the latter in the former's type.

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 [27]. 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)
}
```

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:

```
implicit def sameType[A : Numeric] =
```

Listing 1.16. Adhoc 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))
}
```

```
  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 `~>` type takes two parameters, saying how to obtain a value of the target type from each operand. Here is the definition of the `~>` type:

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

The following listing demonstrates the homogeneous syntax to perform an addition on different data types and the operand coercion between heterogeneous types:

```
def main(i: Rep[Int], j: Rep[Int],
         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 Implementation discussion

We implemented our language as a compiled embedded DSL in Scala. The embedding gave us the opportunity to reuse Scala's infrastructure: we didn't have to write a parser and, more importantly, we could reuse Scala's type system. Furthermore, staging gave us the opportunity to generate efficient code.

Code generation using LMS is usually described as a two step process: a program first evaluates to an intermediate representation and then the final program's code is generated from the intermediate representation. We think we can consider a third step, occurring before the program evaluation: the compilation.

Indeed, the code generation process consists in three steps giving DSL authors as many opportunities to implement their language features. First, features like type coercion and *adhoc* 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 State monad DSL builds an imperative sequence of assignments on a variable. 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.

Each of these steps takes out some information on the program.

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 is useful 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 Validation and comparison with other approaches

Our DSL aims to give developers a safer and more expressive way to write the client-side code. We focused on the main tasks of RIAs development: handling events, asynchronous programming and DOM definition.

4.1 Observed benefits of the use of our DSL

Using our DSL, we were able to divide by two the length of some parts of code (DOM definition and `null` reference handling). The size of code parts dealing with events did not change but they gain type safety. Finally, code handling asynchronous computations were easier to modularize.

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 will be available.

4.2 JavaScript libraries

The most popular JavaScript library, jQuery [28], used by more than 40% of the top million sites[8], and Q.js[9] provide an API turning asynchronous computations into first-class citizen. However, the callback-hell problem can't always be avoided due to the absence of a special syntactic sugar for continuation passing style.

jQuery handles the `null` references problem at runtime 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.

[8] http://trends.builtwith.com/javascript

[9] https://github.com/kriskowal/q

4.3 GWT, Roy, Kotlin

GWT also has a type safe API to handle events.

Roy solves the callback-hell problem.

4.4 Template engines

Templates written using our HTML templating DSL are about two times shorter than templates written using the native JavaScript API. Compared to usual template engines our solution has no runtime overhead, does not require a separated compilation step and provides a flexible type safe expression language.

Furthermore, templates can be shared between client and server sides. The only system, which we are aware of, that allows developers to share their HTML templates on both server and client sides in a statically typed way and with a convenient expression language is Opa. In Opa they made this possible because HTML templating is part of the Opa language itself. If they wanted to support the sharing of another concept such as form field validation rules they would need to introduce this concept as a part of their Opa language.

On the other hand, in our case introducing the support of form field validation as a DSL usable on both server and client sides would be as simple as defining an API and its translation into both Scala and JavaScript. We wouldn't need to change the Scala language itself.

(TODO Make this result more prominent)

5 Conclusion and perspectives

Writing RIAs requires large JavaScript code bases, however this language has several inadequacies for this purpose.

We provide a safer and more expressive language than JavaScript and its native APIs to write RIAs.

More code sharing.

6 Acknowledgements

We thank Tiark Rompf for his precious feedback.

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. B. C. Oliveira, A. Moors, and M. Odersky, "Type classes as objects and implicits," *SIGPLAN Not.*, vol. 45, pp. 341–360, Oct. 2010.
18. T. Hoare, "Null references: The billion dollar mistake," *Presentation at QCon London*, 2009.
19. 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.
20. P. Grogono and P. Chalin, "Copying, sharing, and aliasing," in *In Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94*, 1994.
21. F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for class immutability," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 61–70, ACM, 2011.
22. P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, (New York, NY, USA), pp. 1–14, ACM, 1992.
23. N. Zakas, *Maintainable JavaScript*. O'Reilly Media, 2012.
24. H. Abelson and G. Sussman, "Structure and interpretation of computer programs," 1983.
25. 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.
26. M. Odersky, "Poor man's Type Classes," in *Presentation at the meeting of IFIP WG*, July 2006.
27. 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.

28. B. Bibeault and Y. Kats, *jQuery in Action*. Dreamtech Press, 2008.