# Statically typed Web programming with JavaScript

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 base is challenging because this language has several inadequacies (no static typing, no module system, no *adhoc* polymorphism support, 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, no support of *adhoc* polymorphism, *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

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

to be 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 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 and discusses the implementation of the following contributions:

- We expose native JavaScript APIs for DOM manipulation and event handling as statically typed APIs while reducing their syntactic noise ;

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

- We define a high-level abstraction to handle program's state changes and an abstraction turning asynchronous computations into first-class values. Then we show how to exploit staging to translate these abstractions into JavaScript with a minimal overhead ;
- We use Scala's advanced type-level constructs to elegantly express type coercion constraints on operands in arithmetic operations and to achieve *adhoc* polymorphism with no overhead at all in the generated code ;
- Our code generation system is flexible enough to handle cross-browsers incompatibilities and, unlike other approaches, to allow the sharing of concepts between client-side and server-side even if these concepts are translated into code using external APIs.

The next section reviews the framework used to define compiled embedded DSLs in Scala, sections 3 to 6 present our contributions and section 7 concludes.

## 2 Introduction to Lightweight Modular Staging

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. The bindings between stages is 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 1.1 and 1.2 show the JavaScript and Scala generated code for the `inc` function:

**Listing 1.1.** JavaScript code generation

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

**Listing 1.2.** 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 Improving native JavaScript APIs

This section shows how we built DSLs improving either the expressivity or the safety or both of the equivalent native JavaScript API.

### 3.1 Events

To handle events using the native JavaScript API you have to attach an event handler for a type of events on an object. An event handler is just a function taking the event as a parameter. Consider the following code printing keys hits and mouse clicks in the console:

**Listing 1.3.** Native JavaScript API to handle events

```
window.addEventListener("keyup", function (e) {
  console.log(e.key);
});
window.addEventListener("click", function (e) {
  console.log(e.button);
});
```

We want to "translate" the above API in Scala in a statically typed way without adding verbosity or complexity. 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 `MouseEvent` contains a `button` field returning the value of the mouse button involved in the event. For instance in the above listing, the first handler processs `KeyboardEvent` values because it listens to `keyup` 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: KeyboardEvent =>
  println(e.key)
}
```

```
window.onClick { e: MouseEvent =>
  println(e.button)
}
```

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:

```
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` member. We can then define the `keyup` and `click` events as follows:

```
object KeyUp extends EventDef { type Type = KeyboardEvent }
object Click extends EventDef { type Type = MouseEvent }
```

The following Scala listing is equivalent to the JavaScript listing 1.3 and is completely typesafe: if the user misspells the event name or tries to use an undefined field on an event his program won't compile.

```
window.on(KeyUp) { e =>
  println(e.key)
}
window.on(Click) { e =>
  println(e.button)
}
```

### 3.2 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, but the native API for creating DOM is very verbose, making the code hard to reason about. For instance the following listing shows how to build a `div` element containing a greeting message:

**Listing 1.4.** DOM fragment creation using the native API

```
var hello = function (name) {
  var root = document.createElement("div");
  root.setAttribute("class", "greeting");
  var helloT = document.createTextNode("Hello, ");
  var recipient = document.createElement("strong");
  var recipientT = document.createTextNode(name.toUpperCase());
  recipient.appendChild(recipientT);
  root.appendChild(helloT);
  root.appendChild(recipient);
  return root
};
```

Calling `hello("Obama")` produces a DOM fragment equivalent to the following HTML:

```
<div class="greeting">
  Hello, <strong>OBAMA</strong>
</div>
```

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

```
var hello = function (name) {
  return '<div class="greeting">' +
           'Hello, <strong>'+name.toUpperCase()+'</strong>' +
         '</div>'
};
```

This implementation may be more readable, however the code is wrong: if we call `hello("<me>")` the brackets won't be escaped and will produce a `ME` tag, which is not the expected behavior. So, although this way reads better it's not less error prone.

HTML template engines like Mustache[6] or Closure Templates[7] aim to solve these problems by providing a convenient syntax to describe the HTML structure and allowing the insertion of dynamic expressions in safe way. However, template engines require templates to be compiled, so they add an extra step in the build chain. Some template engines can be used on both client-side and server-side but they have severe limitations (*e.g.* Closure Template has no expression language) and are not statically typed. We'll show in section 6 how we solved this problem.

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 and requires no extra compilation step.

Since 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 1.5.** DOM definition DSL

```
def hello(name: Rep[String]) =
  tag("div", "class"->"greeting") {
    text("Hello, "),
    tag("strong") { text(name.toUpperCase) }
  }
```

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

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

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.

## 4 High-level abstractions without performance penalty

This section shows how we used staging to remove the overhead of high-level DSLs.

### 4.1 Null values

Null values are a known source of problems in programming languages [17,18]. For example, consider the following typical code finding a particular widget in the page and a then particular button in the widget:

**Listing 1.6.** 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 1.7.** 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) { ... });
  }
}
```

The most popular JavaScript library, jQuery [19], used by more than 40% of the top million sites[8] handles this 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. Rewriting the above code using jQuery gives the following listing:

```
var loginWidget = $("div.login");
var loginButton = loginWidget.find("button.submit");
loginButton.on("click", function (e) { ... });
```

We can see that it has as few noise as in the listing 1.6 but it has the safety of the listing 1.7. However, the checking involves a slight performance penalty since each result is wrapped and then unwrapped for the subsequent method invocation.

Our solution has both the safety and performance of the listing 1.7 and the expressiveness of the listing 1.6. We encode the potential emptiness of a value of type `Rep[A]`

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

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 1.7:

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

### 4.2 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 1.8.** 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 1.9.** 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 1.10.** 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 1.11.** 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 1.9 and 1.10 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 1.11 compiles to code equivalent to the listing 1.8.

### 4.3 Future monad

RIAs perform Ajax requests to asynchronously fetch data and then process them in a callback (inversion of control). Dealing with asynchronous code is hard because of the well known "callback hell". Kossakowski *et al.* proposed a way to *un-invert* the control of callback-based APIs using continuations so developers don't really see that their code is asynchronous. For example, consider the following code:

**Listing 1.12.** Asynchronous computations using continuations

```
sleep(1000)
println("Hello")
sleep(1000)
println(", World!")
```

The code looks synchronous however the `sleep` calls are asynchronous: they translate to the following JavaScript:

**Listing 1.13.** Generated code for listing 1.12

```
window.setTimeout(function () {
  console.log("Hello");
  window.setTimeout(function () {
    console.log(", World!");
  }, 1000);
}, 1000);
```

Actually, the listing 1.12 does not compile: at some point developers have to think of the asynchronous translation of their code to delimit the end of the callback generated by the code translation. Consider these two variants:

**Listing 1.14.** Version 1

```
reset {
  sleep(1000)
  println("Hello")
  sleep(1000)
  println(", World!")
  println(42)
}
```

**Listing 1.15.** Version 2

```
reset {
  sleep(1000)
  println("Hello")
  sleep(1000)
  println(", World!")
}
println(42)
```

The two above version behave differently: in listing 1.14 the last `println` statement is included in the callback while it is not the case in listing 1.15. So this last listing will print `42` immediately. Sometimes the enclosing `reset` may be hidden in a function call:

**Listing 1.16.** Ambiguous reading

```
reset {
  foo {
    sleep(1000)
    println("A")
  }
  println("B")
}
```

If `foo` already contains a call to `reset`, the above code will print `BA` instead of `AB` as we may expect.

Because asynchronous computations may not be reflected explicitly in the types and because developers have to look backward to the enclosing `reset` and then forward to see what part of their code block will be captured by the continuation, we think this programming model may be confusing.

In this paper we propose a DSL that explicitly reflects the asynchronous nature of computations in their type and provides methods turning them into first-class citizen. Our 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 ;

The `map` and `flatMap` methods allow asynchronous values to be transformed and passed around, thus turning them into first-class citizen.

Because callbacks are explicitly passed to the `foreach`, `map` and `flatMap` methods, there is no risk of ambiguity such as the one of listing 1.16.

The listing 1.15 can be re-written as follows with our DSL:

```
for {
  _ <- sleep(1000)
  _ <- future(println("Hello"))
  _ <- sleep(1000)
  _ <- future(println(", World!"))
} ()
println(42)
```

The **`for`** notation can intuitively be thought of as a sequencing notation: whenever the first `sleep` call ends, the next statement will be called, and so on. The `println(42)` statement which is outside of the sequence won't be part of it, as expected.

The value of the **for** notation is more evident when asynchronous computations return values:

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

The above listing produces the following code (we renamed some identifiers for the sake of readability):

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

The `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.

```
def f(aF: Rep[Future[String]]) =
  for (a <- aF) yield a.toUpperCase
def g(aF: Rep[Future[String]]) =
  for (a <- aF) yield a.toLowerCase

def fAndG(aF: Rep[Future[String]]) = for {
  u <- f(aF)
  l <- g(aF)
} yield u + l
```

(TODO Explain the value of the above listing...)

## 5 Type-level abstractions

This section shows how we encoded some abstractions at the type level to increase the expressivity of our DSL with no overhead at all in the generated code.

### 5.1 Arithmetic operation operands 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 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 [23]. 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:

```
implicit class NumericOps(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] =
  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)
  println(x + y)
  // i is promoted to Rep[Double], n has type Rep[Double]
  val n = i + y
  // j is promoted to Rep[Double], m has type Rep[Double]
  val m = x + j
  println(n + m)
}
```

### 5.2 *Adhoc* polymorphism

Consider the following listing showing how to define a `show` method, returning a text describing the object on which the method is called, on a `User` and a `Point` data types:

```
// Type 'User'
var User = function (name, age) {
  this.name = name;
  this.age = age;
};

// Type 'Point'
var Point = function (x, y) {
  this.x = x;
  this.y = y;
};

// Implementation of 'show' for a 'User'
User.prototype.show = function () {
  return this.name + " (" + this.age + ")"
};

// Implementation of 'show' for a 'Point'
Point.prototype.show = function () {
  return "(" + this.x + ", " + this.y + ")"
};

// Main program
var main = function (user, point) {
  console.log(user.show());
  console.log(point.show());
};
```

We define the `show` method on the prototypes of the target data types `User` and `Point`, so when the main program calls the `show` method on an object it is automati-

cally dispatched to the right implementation. However, modifying object prototypes is considered to be a bad practice [24]. 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 [25], but this solution is painful for developers and incurs a performance overhead.

Kossakowski *et al.* showed how to use Scala traits to have *adhoc* polymorphism without having to play directly with JavaScript prototypes. However their solution suffers from some limitations: it doesn't support retroactive extension, it does not track the side-effects potentially performed by a method call (so it prevents some optimizations to be applied by LMS), it is limited to single inheritance (and even single interface inheritance) and does not support constructor parameters.

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 [26,27,28].

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

// Polymorphic function
def show[A : Show](a: Rep[A]) = implicitly[Show[A]].show(a)

// Type `User`
type User = Record { val name: String; val age: Int }
// Implementation of Show for a User
implicit val showUser =
  Show[User](user => user.name + " (" + user.age + " years)")

// Type `Point`
type Point = Record { val x: Double; val y: Double }
// Implementation of Show for a Point
implicit val showPint =
  Show[Point](point => "(" + point.x + ", " + point.y + ")")

// Main program
def main(user: Rep[User], point: Rep[Point]) = {
  println(show(user))
  println(show(point))
}
```

The generated code looks like the following:

```
var main = function (user, point) {
  var x0 = showUser(user);
  console.log(x0);
  var x1 = showPoint(point);
  console.log(x1);
};
var showUser = function (user) {
  var x2 = user.name + "(";
  var x3 = x2 + user.age;
  var x4 = x3 + " years)";
```

```
  return x4
};
var showPoint = function (point) {
  var x5 = "(" + point.x;
  var x6 = x5 + ", ";
  var x7 = x6 + point.y;
  var x8 = x7 + ")";
  return x8
};
```

(TODO Conclusion)

# 6 Modular code generation and aspects composition

## 6.1 Reuse the DOM creation DSL on server-side

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. We want to share HTML templates on both server and client sides to get consistency in the rendering and to avoid to duplicate the markup structure.

A few existing template engines can be used on both server and client sides. They provide a convenient syntax to describe a HTML structure in which you can inject dynamic values. However, these template engines are limited because they're not statically typed and they don't provide a convenient expression language for the dynamic values. These two problems are in fact two aspects of a same problem: if we write the expression `xs.size == 0` in a template, how should it 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`). So 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). That's probably for this reason that existing template engines usable on server and client sides have no expression language at all: the dynamic content can only be provided as a map of key-value pairs and no operation can be done on them.

Our DSL presented in section 3.2 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 1.5 produces the following Scala code usable on server-side (the generated code for client-side is roughly equivalent to the listing 1.4):

```
def hello(name: String) = {
  val x0 = {xml.Text("Hello, ")}
```

```
  val x1 = name.toUpperCase
  val x2 = {xml.Text(x1)}
  val x3 = <strong>{x2}</strong>
  val x4 = <div class="greeting">{List(x0, x3)}</div>
  x4
}
```

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.

### 6.2 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. Developers 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. For example, the `forEach` method on JavaScripts arrays was not supported by Internet Explorer before version 9. As some users may still have old versions of Internet Explorer we want to send them a specialized version of JavaScript files to handle this incompatibility. We can do that by extending the code generator for the `List` operations to specialize the generation of the `ListForeach` node to generate a JavaScript **`for`** loop instead of a `forEeach` call:

```
trait JSGenListIE extends JSGenList {
 import IR._
 override def emitNode(sym: Sym[Any], rhs: Def[Any]) =
  rhs match {
   case ListForeach(l, x, b) =>
    var i = fresh[Int]
    stream.println("for (var "+quote(i)+" = 0 ; "+
                         quote(i)+" < "+quote(l)+".length ; "+
                         quote(i)+"++) {")
    emitValDef(x, quote(l)+"["+quote(i)+"]")
    emitBlock(b)
    stream.println("}")
   case _ => super.emitNode(sym, rhs)
  }
}
```

Internet Explorer suffers from some others incompatibilities, *e.g.* its events API is different of the standard. On the same way we can define a `JSGenDomIE` trait generating code targeting the Internet Explorer's events API for the `Dom` DSL.

Then, we can pack together all the traits handling Internet Explorer's incompatibilities in a `IESupport` trait:

```
trait IESupport extends JSGenListIE with JSGenDomIE {
  val IR: ListOpsExp with DomExp
}
```

And, finally, we can mix this `IESupport` trait with any JavaScript code generator to target Internet Explorer:

```
val jsGen = new JSGen with IESupport {
  val IR: prog.type = prog
}
```

## 7 Discussion

We exposed native JavaScript APIs as statically typed APIs while decreasing their verbosity.

## 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. Bibeault and Y. Kats, *jQuery in Action*. Dreamtech Press, 2008.
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. 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.
24. N. Zakas, *Maintainable JavaScript*. O'Reilly Media, 2012.
25. H. Abelson and G. Sussman, "Structure and interpretation of computer programs," 1983.
26. 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.
27. M. Odersky, "Poor man's Type Classes," in *Presentation at the meeting of IFIP WG*, July 2006.
28. B. C. Oliveira, A. Moors, and M. Odersky, "Type classes as objects and implicits," *SIGPLAN Not.*, vol. 45, pp. 341–360, Oct. 2010.