Using Path-Dependent Types to Build Safe JavaScript Foreign Function Interfaces

Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel

IRISA, Université de Rennes 1, France. {first}.{last}@irisa.fr

Abstract. Several programming languages can target JavaScript as a back-end, giving developers programming language features that are absent from JavaScript, such as static typing. However, the Web browser APIs, which are needed to interact with a Web page, are designed for JavaScript, making it challenging to expose them in a statically typed language. Indeed, existing statically typed languages exposing Web browser APIs either break type safety or give developers less control than if they were using JavaScript. This article shows how to expose Web browser APIs in Scala in a type safe way while keeping the same level of control as with native APIs by using path-dependent types and functional dependencies. We validate this approach in designing safe and concise foreign function interfaces between Scala and JavaScript for DOM events handling and DOM manipulation. We compared this approach to common frameworks such as GWT, Fay, Kotlin, Dart and SharpKit.

Keywords: Path-Dependent Types, JavaScript, Scala, Foreign Function Interface

1 Introduction

Web applications are attractive because they require no installation or deployment step on clients and enable large scale collaborative experiences. Besides, now they can be executed efficiently [**?**] on top of modern web-browsers. However, writing large Web applications is known to be difficult [**?**,**?**]. One challenge comes from the fact that the JavaScript programming language – which is currently the only action language natively supported by almost all Web clients – lacks of constructs making large code bases maintainable (*e.g.*static typing, first-class modules).

One solution consists in considering JavaScript as an assembly language[1] and generating JavaScript from compilers of full-featured and cutting-edge programming languages. Incidentally, an increasing number of programming languages or compiler back-ends can generate JavaScript code (*e.g.*Java/GWT [**?**], SharpKit[2], Dart [**?**], Kotlin[3], ClojureScript [**?**], Fay[4], Haxe [**?**], Opa[5]). However, compiling

[1] *cf.* http://asmjs.org/
[2] http://sharpkit.net
[3] http://kotlin.jetbrains.org/
[4] http://fay-lang.org/
[5] http://opalang.org/

to JavaScript is not enough. Developers also need the Web browser programming environment: they need to interact with the Web page, to build DOM fragments, to listen to user events, *etc.*. A Foreign Function Interface (FFI) mechanism could be used to make browser's APIs available to the developers. However, JavaScript APIs are not statically typed and make a heavy use of overloading, making them hard to expose in a statically typed language.

Indeed, existing statically typed languages compiling to JavaScript often expose weaker types than they should. For instance, the function `createElement` is polymorphic in its return type: it can return a `DivElement` as well as an `InputElement`, among others, but the Dart, Fay, SharpKit and Kotlin APIs return the super-type of all the possible values, namely the `Element` type. As a consequence, developers need to explicitly down-cast the value they get, which is a tedious and error prone task.

Some other languages try to workaround this problem by using overloading instead of polymorphism. For instance, HaXe provides functions `createDivElement`, `createInputElement`, which return a `DivElement` and an `InputElement`, respectively. Besides requiring a higher effort to implement, this solution also reduces the control level of users: by being statically resolved, the element type can not anymore be passed as a parameter.

It turns out that most of the existing statically typed languages compiling to JavaScript either loose control or loose type safety when they expose Web browser's APIs. How to give developers the same level of control as if they were using the native Web APIs, but in a statically typed and convenient way?

In this paper we present several ways to integrate Web browser's APIs as statically typed APIs that are safe and give developers the same control level as if they were using the native APIs. We achieve this by using advanced features of type systems like dependent types and functional dependencies. We validate this approach in designing safe FFI between Scala and JavaScript for DOM events handling and DOM manipulation. We compared this approach to common frameworks such as GWT, Fay, Kotlin, Dart and SharpKit.

The outline of this paper is the following. Section **??** presents several motivating example and gives a background about path-dependent type. Section **??** presents Section **??** validates our ideas through the design of FFI for DOM events handling and DOM manipulation in Scala and compare these FFIs w.r.t. the existing FFIs for these features within frameworks such as GWT, Fay, Kotlin, Dart and SharpKit. Section **??** discusses related work, Section **??** concludes this paper and highlights future work.

2 Background Material

2.1 Foreign Function Interface

A FFI is a mechanism that allows a programmer to link one programming language program to programs written in another language. Such mechanisms exist in lots of languages: Ada, Java, Haskell, Lisp or GWT. Despite the name, FFIs

are not necessarily restricted to function calls; many FFIs permit method calls on objects; and some even permit migration of non-trivial datatypes and/or objects across the language boundary. Nevertheless, the challenging tasks come when you want to: (i) manage shared garbage collection policies, (ii) handle the conversion of complex datatype to map from one environment to another and (iii) handle the design of safe foreign function interfaces regarding typing system constraints that can differ from one environment to another. This last issue is the one which is targeted in this paper for the design of FFI between a statically typed language such as Scala and a dynamically typed language such as JavaScript.

2.2

A type system provides us a near mathematical proof that programs that compile successfully obey a set of rules and restrictions and, as such, are bug-free in that respect. For example, after compiling a Java program, developer knows that the it will not do a division of a number with a string and that, if our method foo receives an integer as argument, it will not be called anywhere with an argument of any type other than an integer.

Among all the type theory, dependent types is a term that describes using values as types, the type depends on a value. This means you can encode some properties of an object in its type. Once you do that, it allows a type checker (a compiler) to verify that the object (type) is used in a suitable way. Scala defines a concept of path-dependent type. Scala has the concept of abstract types: types that, just like abstract methods, must be defined by subclasses. Objects can have a type as members. As the term path-dependent type says, the type depends on the path: in general, different paths give rise to different types. In the example of Figure 1.1[6], we show how the size of the list can be used as a type information. We define a function row that convert a list of element into a printed html array, we define a function html that takes a list of size N as a first parameter and a list of list of size N as a second parameter. We use the path dependent type to create these type lists: *Sized*. Consequently if we use this function *html* with a list of three elements (*headers*) and a list of list of 3 elements (*rows*), the program compiles and runs correctly. If we use this function *html* with a list of four elements (*extendedHeaders*) and a list of list of 3 elements (*rows*), it does not compiled.

More advanced type systems, such as the one featured in Scala, increase our confidence in the systems we develop and help us reduce the number of bugs before deployment. Path-dependent types and dependent method types play a crucial role for attempts to encode information into types that is typically only known at runtime. It is important to note that these important validations don't require extra effort from developers -– no need to manually write tests or to run third-party tools — and have no runtime performance impact, which is always a key concern when developing online systems. The goal of this paper is to

[6] highly inspired from https://github.com/milessabin/shapeless#sized-types

```
//functions definition
def row(cols : Seq[String])
        = cols.mkString("<td>", "</td> <td>", "</td>")
def html[N <: Nat]
        (hdrs : Sized[Seq[String], N],
        rows : List[Sized[Seq[String], N]])
                = row(hdrs) :: rows.map(row(_))

//Example1 that compiles
val headers = Sized("Title", "Studio", "Release year")
val rows = List(
        Sized("Ice Age: Dawn of the Dinosaurs", "Fox","2009"),
        Sized("Finding Nemo","Disney Pixar","2003")
)
// headers and rows statically known to have the name number of columns
val formatted = html(headers, rows)                    // Compiles

//Sample 2 that does not compile
// extendedHeaders has the wrong number of columns for rows
val extendedHeaders = Sized("Title", "Author", "Release year", "Runtime")
val badFormatted = html(extendedHeaders, rows)
```

Listing 1.1. Path dependent type example

highlight the benefits of putting the Scala type system for building efficient web applications.

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. Let us trying to come back on some examples of Web API that can no be easily typed correctly without the use of advanced type system.

2.3 DOM Creation

```
var div = document.createElement('div');
var input = document.createElement('input');
```

`div` has type `DivElement` while `input` has type `InputElement`. These types have different methods. As stated in the introduction, most statically typed language have an API that returns an `Element`, which is the least upper bound of the possible types returned by `createElement`, forcing users to explicitly down-cast the result to the expected type, thus loosing type safety.

Alternatively, some languages use overloading.

Consider the following function creating an element and giving it a class name:

```
var create = function (elementType, className) {
  val el = document.createElement(elementType);
  el.className = className;
  return el
};
create('input', 'field').focus();
create('img', 'figure').src = 'http://google.com/logo.png';
```

How to type check the above code? The problem is that the return type of the `create` function depends on its first parameter.

Note that a possible solution in Java could be the following:

```
class ElementName<E> {}
ElementName<InputElement> Input = new ElementName<InputElement>();
ElementName<ImageElement> Img = new ElementName<ImageElement>();

<E> E create(ElementName<E> type, String className) {
  ...
}

create(Input, "field").focus();
create(Img, "figure").setSrc("http://google.com/logo.png");
```

However, the type parameter `E` has to be filled at use-site. If the compiler can infer it, that's fine.

2.4 DOM Events

A similar issue applies to the DOM events API.

```
class EventName<E> {}
EventName<ClickEvent> Click = new EventName<ClickEvent>();

void on(EventName<E> type, Callback<E> callback) { ... }

Callback<ClickEvent> incrementCounter = new Callback<ClickEvent>() {
  @Override
  void execute(ClickEvent event) {
    ...
  }
}

on(Click, incrementCounter);

<E> Stream<E> stream(EventName<E> type) {
  return new Stream<E>() {
    @Override
    void subscribe(Observer<E> observer) {
      on(type, new Callback<E> {
        @Override
        void execute(E event) {
          observer.publish(event);
        }
      });
    }
  }
}
```

Again, the type parameter `E` has to be filled at use-site.

2.5 Selectors

The problem with selectors is slightly different.

```
var input = document.querySelector('input');
var el = document.querySelector('#content');
```

`input` can not have another type than `InputElement`, but `el` can be an element of any type (depending on which element has the id `content`). Return type can not *always* be inferred from the parameters. In this case, down-casting is impossible to avoid, but how to make it optional, and how to make it less risky?

We want to be able to write the following code:

```
val input = document.find(Input) // input has type InputElement
val el = document.find("#content") // el has type Element
val img = document.find[Img](".figure") // note that the type Img is explicitly applied in this c
                                        // but was automatically inferred in previous cases
val error = document.find[Int]("div") // Does not compile: querySelector can not return a number
```

2.6 dependent types

3 Js-scala: A DSL for web programming design using LMS

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 LMS 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 1).

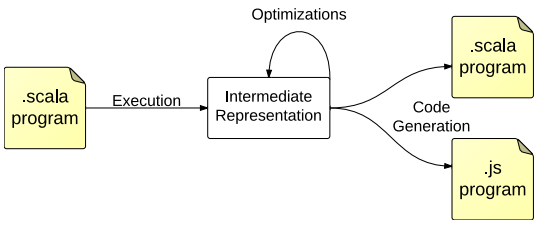

Fig. 1. Compilation of a program using LMS. An initial Scala program using embedded DSLs (on the left) evaluates to an intermediate representation from which the final program's code is generated (on the right).

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:

```
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 1.2 and 1.3 show the JavaScript and Scala generated code for the `inc` function:

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

Listing 1.2. JavaScript code generation

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

Listing 1.3. Scala code generation

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 [**?**,**?**]. For example, consider the following typical JavaScript code finding a particular widget in the page and a then particular button in the widget:

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

Listing 1.4. Unsafe code

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:

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

Listing 1.5. Defensive programming to handle null references

Ja-scala is a DSL that has both the safety and performance of listing 1.5 but the expressiveness of listing 1.4. We can get safety by wrapping potentially null values of type `Rep[A]` in a container of type `Rep[Option[A]]` requiring explicit dereferencing, we can get expressiveness by using the Scala `for` notation for dereferencing, and finally we can get performance by generating code that does not actually wraps values in a container but instead checks if they are `null` or not when dereferenced. The wrapping container exists only at the stage-level and is removed during the code generation. Here is a Scala listing that uses our DSL (implementation details are given in section **??**):

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

The evaluation of the above listing produces a graph of statements from which JavaScript code equivalent to listing 1.5 is generated.

4 Contribution

4.1 Events

Path-dependent types to abstract over an event name and its data type.

4.2 Selectors

Less type annotations on DOM queries, less chance to write nonsense casts

4.3 DOM

Path-dependent types to abstract over an element name and its data type.

5 Evaluation

5.1 Events

Other languages either provide loose information about the data type of the listened event (Dart) or give no way to abstract over an event (GWT, Kotlin).

The most popular JavaScript library, jQuery [?], used by more than 40% of the top million sites[7] 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 the operation effectively 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[8] 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 provide advanced type feature to deal with DOM management.

The Roy [?] 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. It has no specific feature to deal with correctly typed FFI.

GWT [?] compiles Java code to JavaScript. It exposes the browser's APIs through type-safe Java APIs and simplifies DOM fragments definition. However, to provide correct type management, GWT provides a verbose API to deal with DOM fragment definition and event handling. New DOM Api can not be easily managed in a concise and safe way as the Java programming language does not provide advanced type feature.

Kotlin[9] 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. In the specific problem of the definition of concise and safe FFIs it has the same issue than GWT.

Opa[10] and Links [?] are languages designed for Web programming. They feature static typing with global type inference, modules and pattern matching. These languages provide a syntactic sugar for writing HTML fragments directly in the code. The server and client parts of an application are written in a single language and the system can decide whether a code fragment will run on server-side, client-side or both. However, these languages are not extensible, so there is no way for developers to customize the compilation process of their abstractions

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

on client and server sides. Furthermore, hiding the code execution distribution gives programmers less control on communication error handling [**?**].

Dart [**?**] 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.

Table 1 gives a synthetic view of the comparison of our work with other mainstream 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.

TODOFay

	jQuery	TypeScript	Roy	GWT	Kotlin	Opa	Dart	js-scala
Type safe event handling				***				***
DOM definition	*			***	**	***	*	***
Static typing		*	**	**	**	**	*	**
Easiness for defining libraries	**	***	***	***	***	***	***	*

Table 1. Support of concise and safe FFI. Zero star means that the feature is not supported at all, three stars mean a strong support.

Several model-driven methodologies for creating web applications have been proposed, including [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. Many of them follow the standard top-down MDE vision in providing abstractions and code generators, nevertheless these cannot be easily customized or they generate only application skeletons.

6 Conclusion and Perspectives