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 [3] on top of modern web-browsers. However, writing large Web applications is known to be difficult [6,7]. 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 [2], SharpKit[2], Dart [4], Kotlin[3], ClojureScript [5], Fay[4], Haxe [1], 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.

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

Why is it difficult to type Web browser's APIs?

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 case
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 Lightweight Modular Staging

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).

6 Conclusion and Perspectives

References

1. N. Cannasse. Using haxe. *The Essential Guide to Open Source Flash Development*, pages 227–244, 2008.
2. P. Chaganti. *Google Web Toolkit: GWT Java Ajax Programming*. Packt Pub Limited, 2007.
3. Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.
4. R. Griffith. The dart programming language for non-programmers-overview. 2011.
5. M. McGranaghan. Clojurescript: Functional programming for javascript platforms. *Internet Computing, IEEE*, 15(6):97–102, 2011.
6. Tommi Mikkonen and Antero Taivalsaari. Web applications - spaghetti code for the 21st century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 319–328, Washington, DC, USA, 2008. IEEE Computer Society.
7. Juan Carlos Preciado, Marino Linaje Trigueros, Fernando Sánchez-Figueroa, and Sara Comai. Necessity of methodologies to model rich internet applications. In *WSE*, pages 7–13. IEEE Computer Society, 2005.