

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

Julien Richard-Foy and Olivier Barais

IRISA, Universite de Rennes, France

Abstract. Statically typed programming languages compiling to JavaScript make it worth considering the benefits of static typing to write Web applications. Nevertheless, at some point these languages need to expose the Web browser dynamically typed native API, which is a challenging task. Indeed, we observe that existing statically typed languages compiling to JavaScript expose the browser API in ways that either are not type safe or give less expression power, and therefore force developers to pick only one of these two properties. This article presents new ways to encode the challenging parts of the Web browser API in static type systems such that both type safety and expression power are preserved. Our first encoding relies on type parameters and can be implemented in most mainstream languages but drags phantom types up to the usage sites. The second encoding does not suffer from this inconvenience but requires dependent types.

1 Introduction

We recently observed the emergence of several statically typed programming languages compiling to JavaScript (*e.g.* Java/GWT [9], Dart [8], TypeScript [7], Kotlin¹, Opa², SharpKit³, Haxe [2], Scala [6], Idris [1], Elm [5]). These languages make it possible to benefit from static typing (*e.g.* better refactoring support in IDE, earlier error detection) to write Web applications, usually written in the dynamically typed language JavaScript. Nevertheless, at some point developers need a way to interface with the underlying Web browser dynamically typed native API using a *foreign function interface* mechanism.

We observe that, despite these languages are statically typed, their integration of the browser API either is not type safe or gives less expression power. Indeed, integrating an API designed for a dynamically typed language into a statically typed language can be challenging. For instance, the `createElement` function return type depends on the value of its parameter: `createElement('div')` returns a `DivElement`, `createElement('input')` returns an `InputElement`, *etc.*

Most of the aforementioned languages expose this function by making it return an `Element`, the least upper bound of the types of all the possible returned values, thus losing type information and requiring users to explicitly

¹ <http://kotlin.jetbrains.org>

² <http://opalang.org>

³ <http://sharpkit.net>

downcast the returned value to its expected more precise type. Another way to expose this function consists in exposing several functions, each one fixing the value of the initial parameter along with its return type: `createDivElement`, `createInputElement`, *etc.* are parameterless functions returning a `DivElement` and an `InputElement`, respectively. This encoding forces to hard-code the name of the element to create: it can not anymore be a parameter. In summary, the first solution is not type safe and the second solution reduces the expression power.

This paper reviews some common functions of the browser API, identifies the patterns that are difficult to encode in static type systems and shows new ways to encode them such that both type safety and expression power are preserved. We find that type parameters are sufficient to achieve this goal and that *path-dependent types* provide an even more convenient encoding for the end developers.

The remainder of the paper is organized as follows. The next section reviews the most common functions of the browser API and how they are integrated in statically typed languages. Section 3 shows two ways to improve their integration such that type safety and expressivity are preserved. Section 4 validates our contribution and discusses its limits. Section 5 discusses some related works and section 6 concludes.

2 Background

This section reviews the most commonly used browser functions and presents the different integration strategies currently used by the statically typed programming languages GWT, Dart, TypeScript, Kotlin, Opa, SharpKit, Haxe, Scala, Idris and Elm.

All languages support a foreign function interface mechanism, allowing developers to write JavaScript expressions from their programs. This mechanism is generally untyped and error prone, that's why most languages (TypeScript, Kotlin, Opa, SharpKit, Haxe, Scala and Elm) support a way to define *external* typed interfaces. Most of them also expose the browser API through this way, as described in the next section.

2.1 The browser API and its integration in statically typed languages

The client-side part of the code of a Web application essentially reacts to user events (*e.g.* mouse clicks), triggers actions and updates the document (DOM) according to their effect. Table 2.1 lists the main functions supported by Web browsers according to the Mozilla Developer Network⁴ (we removed the functions that can trivially be encoded in a static type system).

To illustrate the challenges raised by these functions, we present a simple JavaScript program using them and show how it can be implemented in statically typed programming languages according to the different strategies used to

⁴ https://developer.mozilla.org/en-US/docs/DOM/DOM_Reference/Introduction

Name	Description
<code>getElementsByTagName(name)</code>	Find elements by their tag name
<code>getElementById(id)</code>	Find an element by its <code>id</code> attribute
<code>createElement(name)</code>	Create an element
<code>target.addEventListener(name, listener)</code>	React to events

Table 1. Web browsers main functions that are challenging to encode in a static type system

encode these functions. Listing 1 shows the initial JavaScript code of the program. It defines a function `slideshow` that creates a slide show from an array of image URLs. The function returns an image element displaying the first image of the slide show, and each time a user clicks on it with the mouse left button the next image is displayed.

```
function slideshow(sources) {
  var img = document.createElement('img');
  var current = 0;
  img.src = sources[current];
  img.addEventListener('click', function (event) {
    if (event.button == 0) {
      current = (current + 1) % (sources.length - 1);
      img.src = sources[current];
    }
  });
  return img
}
```

Listing 1. JavaScript function creating a slide show from an array of image URLs

The most common way to encode the DOM API in statically typed languages is to follow the standard interface specifications of HTML [17] and DOM [4].

The main challenge comes from the fact that these function return types or parameter types are often too general. Indeed, functions `getElementsByTagName(name)`, `getElementById(id)` and `createElement(name)` can return values of type `DivElement` or `InputElement` or any other subtype of `Element` (their least upper bound). The interface of `Element` is more general and provides less features than its subtypes. For instance, the `ImageElement` type (representing images) has a `src` property that does not exist at the `Element` level. Similarly, the `MouseEvent` type has a `button` property that does not exist at the (more general) `Event` level, used by the function `addEventListener`.

Listing 2 shows a Scala implementation of the `slideshow` program using an API following the standard specifications of HTML and DOM. The listing contains two type casts, needed to use the `src` property on the `img` value and the `button` property on the `event` value, respectively.

```

def slideshow(sources: Array[String]): ImageElement = {
  val img =
    document.createElement("img").asInstanceOf[ImageElement]
  var current = 0
  img.src = sources(current)
  img.addEventListener("click", event => {
    if (event.asInstanceOf[MouseEvent].button == 0) {
      current = (current + 1) % (sources.size - 1)
      img.src = sources(current)
    }
  })
  img
}

```

Listing 2. Scala implementation of `slideshow` using the standard HTML and DOM API

These type casts make the code more fragile and less convenient to read and write. That's why some statically typed languages attempt to provide an API preserving as precise types as possible.

In the case of `getElementById(id)`, the `id` parameter gives no clue on the possible type of the searched element so it is hard to infer more precisely the return type of this function. Hence, most implementations define a return type of `Element`.

However, in the case of `getElementsByTagName(name)` and `createElement(name)`, there is exactly one possible return type for each value of the `name` parameter: *e.g.* `getElementsByTagName('input')` always returns a list of `InputElement` and `createElement('div')` always returns a `DivElement`. This characteristic makes it possible to encode these two functions by defining as many parameterless functions as there are possible tag names, where each function fixes the initial `name` parameter to be one of the possible values and exposes the corresponding specialized return type.

The case of `target.addEventListener(name, listener)` is a bit different. The `name` parameter defines the event to listen to and the `listener` parameter the function to call back each time such an event occurs. Instead of being polymorphic in its return type, it is polymorphic in its `listener` parameter. Nevertheless, a similar property as above holds: there is exactly one possible type for the `listener` parameter for each value of the `name` parameter. For instance, a listener of 'click' events is a function taking a `MouseEvent` parameter, a listener of 'keydown' events is a function taking a `KeyboardEvent` parameter, and so on. The same pattern as above (defining a set of functions fixing the `name` parameter value) can be used to encode this function in statically typed languages.

Listing 3 shows what would our `slideshow` implementation look like using such an encoding. There are two modifications compared to listing 2: we use `document.createImageElement` instead of `document.createElement`, and we use `img.addClickListener` instead of `img.addEventListener`.

```
def slideshow(sources: Array[String]): ImageElement = {
  val img = document.createElement()
  var current = 0
  img.src = sources(current)
  img.addEventListener { event =>
    if (event.button == 0) {
      current = (current + 1) % (sources.size - 1)
      img.src = sources(current)
    }
  }
  img
}
```

Listing 3. Scala implementation of `slideshow` using specialized functions

The `createElement` function takes no parameter and returns a value of type `ImageElement`, and the `addEventListener` function takes as parameter a function that takes a `MouseEvent` value as parameter, ruling the need of type casts out.

In the case of the `addEventListener` function we also encountered a slight variation of the encoding, consisting of defining one general function taking one parameter carrying both the information of the event name and the event listener.

```
img.addEventListener(ClickEventListener { event =>
  // ...
})
```

Listing 4. Implementation of `slideshow` using a general `addEventListener` function taking one parameter containing both the event name and the even listener

Listing 4 shows the relevant changes in our program if we use this encoding. The `addEventListener` function takes one parameter, a `ClickEventListener`, carrying both the name of the event and the event listener code.

Most of the studied languages expose the browser API following the standard specification, but some of them (GWT, Dart, HaXe and Elm) define a modified API getting rid of (or at least reducing) the need for downcasting, following the approaches described above.

2.2 Limitations of existing encoding approaches

We distinguished three approaches to integrate the challenging parts of the browser API into statically typed languages. This section compares these approaches in terms of type safety and expression power.

The first approach, consisting in using the least upper bound of all the possible types has the same expression power as the native browser API, but is not type safe because it sometimes requires developers to explicitly downcast values to their expected specialized type.

The second approach, consisting in defining as many functions as there are possible return types of the encoded function, is type safe but leads to a less general API: each function fixes a parameter value of the encoded function, hence being less general. The limits of this approach are better illustrated when one tries to combine several functions. Consider for instance listing 5 defining a JavaScript function `findAndListenTo` that both finds elements and registers an event listener when a given event occurs on them. Note that the event listener is passed both the event and the element: its type depends on both the tag name and the event name. This function can not be implemented if the general functions `getElementsByTagName` and `addEventListener` are not available. The best that could be done would be to create one function for each combination of tag name and event name, leading to an explosion of the number of functions to implement. Thus, this approach gives less expression power than the native browser API.

```
function findAndListenTo(tagName, eventName, listener) {
  var elements = document.getElementsByTagName(tagName);
  elements.forEach(function (element) {
    element.addEventListener(eventName, function (event) {
      listener(event, element);
    });
  });
}
```

Listing 5. Combination of use of `getElementsByTagName` and `addEventListener`

The third approach, consisting in combining two parameters into one parameter carrying all the required information, is type safe too, but reduces the expression power because it forbids developers to partially apply the function by supplying only one parameter. Consider for instance listing 6 that defines a function `observe` partially applying the `addEventListener` function⁵. Such a function can not be implemented with this approach because the name of the event and the code of the listener can not be decoupled. Thus, this one gives less expression power than the native browser API.

In summary, the current integration of the browser API by statically typed languages compiling to JavaScript is either not type safe or not as expressive as the underlying JavaScript API. Indeed, we showed that our simple `slideshow` program requires type casts if the browser API is exposed according to the

⁵ The code of this function has been taken (and simplified) from the existing functional reactive programming libraries Rx.js [13] and Bacon.js (<http://baconjs.github.io/>).

```

function observe(target, name) {
  return function (listener) {
    target.addEventListener(name, listener);
  }
}

```

Listing 6. Partial application of `addEventListener` parameters

standard specification. We are able to get rid of type casts on this program by using modified browser APIs, but we presented two functions that we were not able to implement using these APIs, showing that they give less expression power than the native API.

This article aims to answer the following questions: is it possible to expose the browser API in statically typed languages in a way that both reduces the need for type casts and preserves the same expression power? What typing mechanisms do we need to achieve this? Would it be convenient to use by end developers?

3 Contribution

In this section we show how we can encode the challenging main functions of the DOM API in a type safe way while keeping the same expression power.

Our listings use the Scala language, though our first solution could be implemented in any language with basic type parameters support, such as Java's *generics*. Our second solution is an improvement of the first one, using *path-dependent types*.

3.1 Parametric Polymorphism

In all the cases where a type `T` involved in a function depends on the value of a parameter `p` of this function (all the aforementioned functions of the DOM API are in this case), we can encode this relationship in the type system using type parameters as follows:

1. Define a parametrized class `P[U]`
2. Set the type of `p` to `P[U]`
3. Use type `U` instead of type `T`
4. Define as many values of type `P[U]` as there are possible values for `p`, each one fixing its `U` type parameter to the corresponding more precise type

Listing 7 shows this approach applied to the `createElement` and `getElementsByTagName` functions which return type depends on their `name` parameter value: a type `ElementName[E]` has been created, the type of the `name` parameter has been set to `ElementName[E]` instead of `String`, and the return type of the function is `E` instead of `Element` (or `Array[E]` instead of `Array[Element]`, in the case of

```

class ElementName[E]

trait Document {
  def createElement[E](name: ElementName[E]): E
  def getElementsByTagName[E](name: ElementName[E]): Array[E]
}

val Input = new ElementName[InputElement]
val Img = new ElementName[ImageElement]
// etc. for each possible element name

```

Listing 7. Encoding of the `createElement` function using type parameters

`getElementsByTagName`). The `ElementName[E]` type encodes the relationship between the name of an element and the type of this element⁶. For instance, we created a value `Input` of type `ElementName[InputElement]`.

```

class EventName[E]

trait EventTarget {
  def addEventListener[E](
    name: EventName[E], callback: E => Unit): Unit
}

val Click = new EventName[MouseEvent]
val KeyUp = new EventName[KeyboardEvent]
// etc. for each possible event name

```

Listing 8. Encoding of the `addEventListener` function using type parameters

Listing 8 shows the encoding of the `addEventListener` function. The `EventName[E]` type represents the name of an event which type is `E`. For instance, `Click` is a value of type `EventName[MouseEvent]`: when a user adds an event listener to the `Click` event, it fixes to `MouseEvent` the type parameter `E` of the `callback` function passed to `addEventListener`.

Listing 9 illustrates the usage of such an encoding by implementing our `slideshow` program presented in the introduction. Passing the `Img` value as a parameter to the `createElement` function fixes its `E` type parameter to `ImageElement` so the returned value has the most possible precise type and the `src` property can be used on it. Similarly, passing the `Click` value to the `addEventListener` function fixes its `E` type parameter to `MouseEvent`, so the event listener has the most possible precise type and the `button` property can be used on the `event`

⁶ The type parameter `E` is also called a *phantom type* [11] because `ElementName` values never hold a `E` value


```

def slideshow(sources: Array[String]) {
  val img = document.createElement(Img)
  var current = 0
  img.src = sources(current)
  img.addEventListener(Click, event => {
    if (event.button == 0) {
      current = (current + 1) % (sources.length - 1)
      img.src = sources(current)
    }
  })
  img
}

```

Listing 9. Scala implementation of the `slideshow` function using generics

parameter. It is worth noting that there is no type cast in the listing because the browser API is exposed in a way that preserves enough type information.

```

def findAndListenTo[A, B](
  tagName: ElementName[A],
  eventName: EventName[B],
  listener: (A, B) => Unit) = {
  for (element <- document.getElementsByTagName(tagName)) {
    element.addEventListener(eventName, event => {
      listener(event, element)
    })
  }
}

```

Listing 10. Combination of `getElementsByTagName` and `addEventListener` functions encoded using type parameters

Finally, listing 10 and 11 show how the challenging functions of section 2.2, `findAndListenTo` and `observe`, can be implemented with our encoding. They are basically a direct translation from JavaScript syntax to Scala syntax, with additional type annotations. Our encoding is type safe and gives as much expression power as the native API since it is possible to implement exactly the same functions as we are able to implement in plain JavaScript.

However, every function taking an element name or an event name as parameter has its type signature cluttered with phantom types (extra type parameters): the `observe` function takes a phantom type parameter `A` and the `findAndListenTo` function takes two phantom type parameters, `A` and `B`.

```
def observe[A](target: EventTarget, name: EventName[A]) = {
  (listener: A => Unit) => {
    target.addEventListener(name, listener)
  }
}
```

Listing 11. Partial application of `addEventListener` encoded with type parameters

3.2 Path-Dependent Types

This section shows how we can remove the extra type parameters needed in the previous section by using *path-dependent types* [15]. Essentially, the idea is to model type parameters using *type members*, as suggested in [16].

Usually, in object oriented programming languages, a type `T` can have abstract methods. In Scala it can also have abstract type members. Such a type `T` is abstract and its concrete subtypes have to define their type members (like abstract methods have to be implemented).

```
trait ElementName {
  type Element
}

trait Document {
  def createElement(name: ElementName): name.Element
  def getElementsByTagName(
    name: ElementName): Array[name.Element]
}

object Div extends ElementName {
  type Element = DivElement
}

object Input extends ElementName {
  type Element = InputElement
}

// etc. for each possible element name
```

Listing 12. Encoding of `createElement` using path-dependent types

Listings 12 and 13 show an encoding of `createElement`, `getElementsByTagName` and `addEventListener` in Scala using type members. Now, the `ElementName` type has no type parameter but a type member `Element`. The return type of the `createElement` function is `name.Element`: it refers to the `Element` type member of its `name` parameter. The `Div` and `Input` values illustrate how their corresponding element type is fixed: if one writes `createElement(Input)`, the return type is the `Element` type member of the `Input` value, namely `InputElement`. The same idea

```

trait EventName {
  type Event
}

object Click extends EventName { type Event = MouseEvent }

trait EventTarget {
  def addEventListener(name: EventName)
    (callback: name.Event => Unit): Unit
}

```

Listing 13. Encoding of `addEventListener` using path-dependent types

applies to `EventName` and `addEventListener`: the name of the event fixes the type of the callback.

```

def findAndListenTo(eltName: ElementName, evtName: EventName)
  (listener: (evtName.Event, eltName.Element) => Unit) = {
  for (element <- document.getElementsByTagName(eltName) {
    element.addEventListener(evtName) { event =>
      listener(event, element)
    }
  }
}

```

Listing 14. Combination of `getElementsByTagName` and `addEventListener` using path-dependent types

```

def observe(target: EventTarget, name: EventName) =
  (listener: (name.Event => Unit)) => {
    target.addEventListener(name)(listener)
  }

```

Listing 15. Partial application of `addEventListener` using path-dependent types

The implementation of the `slideshow` function with this encoding is exactly the same as with the previous approach using generics. However, functions `findAndListenTo` and `observe` can be implemented more straightforwardly, as shown by listings 14 and 15, respectively.

With this encoding, the functions using event names or element names are not anymore cluttered with phantom types, and type safety is still preserved.

4 Validation

4.1 Implementation in js-scala

We implemented our encoding in js-scala [10], a Scala library providing composable JavaScript code generators⁷. On top of that we implemented various samples, including non trivial ones like a realtime chat application and a poll application.

We showed in this paper that our encoding ascribes types as precise as possible (our `slideshow` program is free of type casts) while being expressive enough to implement the challenging `findAndListenTo` and `observe` functions that were impossible to implement with other approaches.

4.2 Convenience to use for end developers

Statically typed languages can be criticized for the verbosity of the information they add compared to dynamically typed languages [14]. In our case, what is the price to pay to get accurate type ascriptions?

The first encoding, using type parameters, can be implemented in most programming languages because it only requires a basic support of type parameters (for instance Java, Dart, TypeScript, Kotlin, HaXe, Opa, Idris and Elm can implement it). However this encoding leads to cluttered type signatures and forces functions parametrized by event or element names to also take phantom type parameters.

However, the second encoding, using type members, leads to type signatures that are not more verbose than those of the standard specifications of the HTML and DOM APIs, so we argue that there is no price to pay. This encoding can only be implemented in language supporting type members or dependent types (Scala and Idris), though.

4.3 Limitations

Our encodings only work with cases where a polymorphic type can be fixed by a value. In our examples, the only one that is not in this case is `getElementById`. Therefore we are not able to type this function more accurately (achieving this would require to support the DOM tree itself in the type system as in [12]).

Our solution is actually slightly less expressive than the JavaScript API: indeed, the value representing the name of an event or an element is not anymore a `String`, so it can not anymore be the result of a `String` manipulation, like *e.g.* a concatenation. Fortunately, this case is uncommon.

⁷ Source code is available at <http://github.com/js-scala>

5 Related Works

The idea of using dependent types to type JavaScript has already been explored by Ravi Chugh *et. al.* [3]. They showed how to make a subset of JavaScript statically typed using a dependent type system. However, their solution requires complex and verbose type annotations to be written by developers.

Sebastien Doreane proposed a way to integrate JavaScript APIs in Scala [6]. His approach allows developers to seamlessly use JavaScript APIs from statically typed Scala code. However, he did not use the features of Scala's type system, as we did, to ascribe more precise types when possible.

TypeScript supports overloading on constant values: the type of the expression `createElement("div")` is statically resolved to `DivElement` by the constant parameter value `"div"`. This solution is type safe and as expressive as the native API but has limited applicability (overload resolution requires parameters to be constant values, so the `findAndListenTo` function would be weakly typed with this approach).

6 Conclusion

We presented two ways to encode dynamically typed browser functions in mainstream statically typed languages like Java and Scala, using type parameters or path-dependent types. Our encodings give more type safety than existing solutions, while keeping the same expression power as the native API.

We argue that, if industry shippers want to write their Web applications in statically typed languages, dependent types are going to be the most desired feature of these languages.

References

1. EDWIN BRADY. Idris, a general purpose dependently typed programming language: Design and implementation.
2. N. Cannasse. Using haxe. *The Essential Guide to Open Source Flash Development*, pages 227–244, 2008.
3. Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. *SIGPLAN Not.*, 47(10):587–606, October 2012.
4. W3C-World Wide Web Consortium et al. Document object model (dom) level 3 core specification. *W3C recommendation*, 2004.
5. Evan Czaplicki. Elm: Concurrent frp for functional guis, 2012.
6. Sébastien Doeraene. Scala.js: Type-Directed Interoperability with Dynamically Typed Languages. Technical report, 2013.
7. Steve Fenton. Typescript for javascript programmers. 2012.
8. R. Griffith. The dart programming language for non-programmers-overview. 2011.
9. Federico Kereki. Web 2.0 development with the Google web toolkit. *Linux J.*, 2009(178), February 2009.

10. Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. JavaScript as an Embedded DSL. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 409–434, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
11. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.
12. BenjaminS. Lerner, Liam Elbert, Jincheng Li, and Shriram Krishnamurthi. Combining Form and Function: Static Types for JQuery Programs. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, page 79–103. Springer Berlin Heidelberg, 2013.
13. Jesse Liberty and Paul Betts. Reactive extensions for javascript. In *Programming Reactive Extensions and LINQ*, pages 111–124. Springer, 2011.
14. Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages.
15. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer Berlin Heidelberg, 2003.
16. Martin Odersky and Matthias Zenger. Scalable component abstractions. In *ACM Sigplan Notices*, volume 40, pages 41–57. ACM, 2005.
17. Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.