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

Julien Richard-Foy and Olivier Barais

IRISA, Universite de Rennes, France

Abstract. Encoding dynamically typed APIs in statically typed languages can be challenging. Incidentally, existing statically typed languages compiling to JavaScript expose its native API in ways that either are not type safe or give less expression power to developers. We present two new ways to encode the challenging parts of the Web browser API such that 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 use 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 [7], Dart [6], TypeScript [5], Kotlin[1], Opa[2], SharpKit[3], Haxe [2], Idris [1], Elm [4]). Developers can use them to write the client-side part of their Web applications, relying on a *foreign function interface* mechanism to call the Web browser native API.

However 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.*

Existing languages expose this function by making it return an `Element`, the least upper bound of the types of all the possible returned values, thus loosing type information and requiring users to explicitly 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 parameter along with its return type: `createDivElement`, `createInputElement`, *etc* would be parameterless functions returning a `DivElement` and an `InputElement`, respectively. This encoding forces to hard-code the name of the element to create: it can

[1] http://kotlin.jetbrains.org
[2] http://opalang.org
[3] http://sharpkit.net

not anymore be a parameter. 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 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 ways to improve their integration. Section 4 validates our contribution. Section 5 discusses some related works and section 6 concludes.

2 Background

This section reviews the most commonly used browser functions and presents the way they are integrated in GWT, Dart, TypeScript, Kotlin, Opa, SharpKit, Haxe, Idris and Elm.

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 the Web browsers according to the Mozilla Developer Network[4] (we removed the functions that can trivially be encoded in a static type system).

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

Table 1. Web browsers main functions

The main challenge comes from the fact that these functions are polymorphic in their return type or in their parameters.

For instance, 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, so it is important for

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

the encoding of these functions in a statically typed language to be as precise 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`. The drawback of this approach is that developers may get a value with a type weaker than they need and may require to explicitly downcast it, thus loosing type safety.

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 each of these 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. Listing 1.1 illustrates such an encoding in the Java language. Note that the listing gives only the type signature of the functions visible to the end developers. The implementation of the functions is not relevant for our purpose.

```
DivElement createDivElement();
InputElement createInputElement();
FormElement createFormElement();
...
```

Listing 1.1. Encoding of the `createElement(name)` function in Java using parameterless functions fixing the initial `name` parameter value

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 a statically typed language, but an alternative encoding could be to define one function taking one parameter carrying both the information of the event name and the event listener. Listing 1.2 shows such an encoding in Java.

Table 2.1 summarizes, for each studied statically typed language, which approach is used to encode each function of the browser API.

```
// --- API Definition
interface EventTarget {
  void addEventListener(EventListener listener);
}

interface EventListener {}

interface ClickEventListener extends EventListener {
  void onClick(ClickEvent event);
}

interface KeyDownEventListener extends EventListener {
  void onKeyDown(KeyboardEvent event);
}

// --- Usage
void logClicks(EventTarget target) {
  target.addEventListener(new ClickEventListener {
    public void onClick(ClickEvent event) {
      console.log(event.button);
    }
  });
}
```

Listing 1.2. Encoding of the `target.addEventListener(name, listener)` function in Java using one paremeter carrying both the information of the event name and the event listener

	`getElementById`	`getElementsByTagName`	`createElement`	`addEventListener`
Java	lub	lub	comb	plop
Dart	lub	lub		
TypeScript				
Haxe	lub	lub		
Opa				
Kotlin	lub	lub		
SharpKit	lub	lub		
Idris				
Elm				

Table 2. Summary of the encodings used by existing statically typed languages

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 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 1.3 defining a JavaScript function that both find 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 could not be implemented if the general `getElementsByTagName` and `addEventListener` functions were not available. The best that could be done would be to create one function for each combination of tag name and event name.

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

Listing 1.3. 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 1.4 that defines a function partially applying the `addEventListener` function[5]. Such a function is impossible to encode using this approach, in a statically typed language.

[FIXME En fait les deux dernieres approches sont deux cas particuliers d'une meme approche, plus generale, qui consiste a fixer la valeur d'un parametre. Je regroupe ou pas ?]

In summary, browser API integration by existing statically typed languages compiling to JavaScript is either not type safe or not as expressive as the underlying JavaScript API.

[5] This pattern is often used by functional reactive programming libraries like Rx.js [10]

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

Listing 1.4. Partial application of `addEventListener` parameters

3 Contribution

In this section we show how we can encode the functions presented previously, in a type safe way while keeping the same expression power.

We first propose a solution in the Java mainstream language, using *generics*, and show how we can improve it using *path-dependent types*, in Scala.

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 aforementionned functions are in this case, excepted `getElementById`), we can encode this relationship in the type system using type parameters as follows:

1. Create a parametrized class `P<U>`
2. Set the type of `p` to `P<U>`
3. Use type `U` instead of type `T`

Listing 1.5 shows this approach applied to the `createElement` function which return type depends on its `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`. The `ElementName<E>` type encodes the relationship between the name of an element and the type of this element[6]. For instance, we created a value `Input` of type `ElementName<InputElement>`. The last two lines shows how this API is used by end developers: passing the `Input` value as parameter to the function fixes the type parameter `E` to `InputElement` so the returned value has the most possible precise type.

`getElementsByTagName` can be encoded in very similar way, and listing 1.6 shows the encoding of the `addEventListener` function.

We show that our encodings support the challenging functions defined in listings 1.3 and 1.4.

Listing 1.7 and 1.8 show how these functions can be implemented. They are basically a direct translation from JavaScript to Java.

[6] The type parameter `E` is also called a *phantom type* [9] because `ElementName` values never hold a `E` value

```
// --- API Definition
class ElementName<E> {}

<E> E createElement(ElementName<E> name);

final ElementName<InputElement> Input = new ElementName<InputElement>();
final ElementName<ImageElement> Img = new ElementName<ImageElement>();

// --- Usage
InputElement input = createElement(Input);
ImageElement img = createElement(Img);
```

Listing 1.5. Encoding of the `createElement` function using type parameters

```
// --- API Definition
class EventName<E> {}

interface EventTarget {
  <E> void addEventListener(EventName<E> name, Function<E, Void> callback);
}

final EventName<MouseEvent> Click = new EventName<MouseEvent>();

// --- Usage
ButtonElement btn = createElement(Button);
btn.addEventListener(Click, e -> console.log(e.button))
```

Listing 1.6. Encoding of the `addEventListener` function using type parameters

```
<A, B> void findAndListenTo(
    ElementName<A> tagName,
    EventName<B> eventName,
    Function2<A, B, Void> listener) {
  for (A element : document.getElementsByTagName(tagName)) {
    element.addEventListener(eventName, event -> listener.apply(event, element));
  }
}
```

Listing 1.7. Combination of `getElementsByTagName` and `addEventListener` functions encoded using type parameters

```
<A> Function<Function<A, Void>, Void> observe(EventTarget target, EventName<A> name)
  return listener -> {
    target.addEventListener(name, listener);
  }
}
```

Listing 1.8. Partial application of `addEventListener` encoded with type parameters

Our encoding is type safe and gives as much expression power as the native API since it makes it 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).

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*. Essentially, the idea is that, instead of using a type parameter, we use a type member. Listings 1.9 and 1.10 show this encoding in Scala. The return type of the `createElement` function is `name.Element`: it refers to the `Element` type member of its `name` parameter.

```
trait ElementName {
  type Element
}
object Div extends ElementName { type Element = DivElement }
object Input extends ElementName { type Element = InputElement }

def createElement(name: ElementName): name.Element
```

Listing 1.9. Encoding of `createElement` using path-dependent types

```
trait EventName {
  type Event
}
object Click extends EventDef { type Event = MouseEvent }

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

Listing 1.10. Encoding of `addEventListener` using path-dependent types

Implementing listings 1.3 and 1.4 using this encoding is straightforward, as shown by listings 1.11 and 1.12, 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.

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

Listing 1.11. Combination of `getElementsByTagName` and `addEventListener` using path-dependent types

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

Listing 1.12. Partial application of `addEventListener` using path-dependent types

4 Validation

4.1 Implementation in js-scala

We implemented our pattern in js-scala [8], a Scala library providing JavaScript code generators[7].

4.2 Limitations

We do not help with `getElementById`.

The name of an element or the name of an event can not anymore be a value resulting of the concatenation of Strings.

5 Related Works

Ravi Chugh *et. al.* [3] showed how to make a subset of JavaScript statically typed using a dependent type system. They require complex type annotations to be written by developers.

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 encoding gives 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.

[7] Source code is available at http://github.com/js-scala

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. Evan Czaplicki. Elm: Concurrent frp for functional guis, 2012.
5. Steve Fenton. Typescript for javascript programmers. 2012.
6. R. Griffith. The dart programming language for non-programmers-overview. 2011.
7. Federico Kereki. Web 2.0 development with the Google web toolkit. *Linux J.*, 2009(178), February 2009.
8. 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.
9. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.
10. Jesse Liberty and Paul Betts. Reactive extensions for javascript. In *Programming Reactive Extensions and LINQ*, pages 111–124. Springer, 2011.