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. We present new ways to encode them in order to preserve type safety and to keep the same expression power.

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 to developers. 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 the `String` parameter it is passed.

This paper reviews some common functions of the browser API, identifies those which are not well encoded in statically typed programming languages and shows new ways to encode them with more type safety while keeping the native API expression power. 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.

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

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 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.

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. Neverthe-

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

```
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

less, 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.

```
// --- 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

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

	`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, expression power and convenience to use from the end developer point of view.

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 a function, is completely type safe but can lead to combinatorial explosion in some situations. Consider for instance listing 2.2 defining a JavaScript function that both creates an element and registers an event listener when a given event occurs on it. Note that the event listener is passed both the event and the element. Encoding such a function in a statically typed language using this approach would require to create one function for each combination of tag name and event name.

```
var createAndListenTo = function (tagName, eventName, listener) {
  var element = document.createElement(tagName);
  element.addEventListener(eventName, function (event) {
    listener(event, element);
  });
};
```

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 2.2 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.

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

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

3 Contribution

In this section we show how we can encode the functions presented previously, in a type safe and convenient way for developers 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 3.1 shows this approach applied to the `createElement` function: we created a type `ElementName<E>`, the `name` parameter is not a `String` but a `ElementName<E>`, and the function returns an `E` instead of an `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>`.

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

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

The last two lines shows how this API is used by end developers: by passing the `Input` value as parameter to the function, it fixes the type parameter `E` to `InputElement` so the returned value has the most possible precise type.

```
// --- 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);
```

`getElementsByTagName` can be encoded in very similar way, and listing 3.1 shows the encoding of the `addEventListener` function.

```
// --- 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))
```

We show that our encodings support the challenging functions defined in listings 2.2 and 2.2.

Listing 3.1 and 3.1 show how these functions can be implemented. They are basically a direct translation from JavaScript to Java.

Our encoding makes it possible to implement exactly the same functions as we are able to implement in plain JavaScript, but with type safety.

However, every function taking an element name or an event name as parameter must have type parameters too, making its type signature harder to read.

```
<A, B> void createAndListenTo(
    ElementName<A> tagName,
    EventName<B> eventName,
    Function2<A, B, Void> listener) {
  A element = document.createElement(tagName);
  element.addEventListener(eventName, event -> listener.apply(event, element));
}
```

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

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 3.2 and 3.2 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
```

Implementing listings 2.2 and 2.2 using this encoding is straightforward, as shown by listings 3.2 and 3.2, respectively.

With this encoding, the functions using event names or element names are not anymore cluttered with type parameters.

4 Validation

4.1 Implementation in js-scala

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

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

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

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


def createAndListenTo(tagName: ElementName, eventName: EventName, listener: (eventNam
  val element = document.createElement(tagName)
  element.addEventListener(eventName)(event => listener(event, element))
}
```

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 *generics* or path-dependent types. Our encoding gives more type safety than existing solutions, while keeping the same expression power and modularity level as the native API.

References

1. EDWIN BRADY. Idris, a general purpose dependently typed programming language: Design and implementation.

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

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.