FFI Is Not Enough. We Need Dependent Types

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

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

Abstract. JavaScript sucks. It is a known fact. Several other programming languages can target JavaScript as a back-end, alleviating developers from the burden of writing and maintaining JavaScript code. 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 either loose control or loose type safety. How to give users the same level of control as if they were using the native Web APIs, but in a statically typed and convenient way? This article shows how dependent type systems can help.

Keywords: Dependent Types

1 Introduction

Web applications are attractive because they require no installation or deployment step on clients and enable large scale collaborative experiences. 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 backends can generate JavaScript code (*e.g.*Java/GWT [**?**], SharpKit[2], Dart [**?**], Kotlin[3], ClojureScript [**?**], Fay[4], Haxe [**?**], Opa[5]).

However, compiling 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 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.

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

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 can achieve this by using advanced features of type systems like dependent types and functional dependencies.

2 Motivating Example

Typical tasks involved in Web applications.

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

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
- Inference-driving macros help inferring more specific types

4.3 DOM

?

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. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
2. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
3. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
4. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
5. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
6. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`