Static Typing & JavaScript Libraries: Towards a More Considerate Relationship*

Benjamin Canou
LIP6 - UMR 7606
Université Pierre et Marie Curie
Sorbonne Universités
4 place Jussieu, 75005 Paris, France
Benjamin.Canou@lip6.fr

Emmanuel Chailloux
LIP6 - UMR 7606
Université Pierre et Marie Curie
Sorbonne Universités
4 place Jussieu, 75005 Paris, France
Emmanuel.Chailloux@lip6.fr

Vincent Botbol
Université Pierre et Marie Curie
Sorbonne Universités
4 place Jussieu, 75005 Paris, France
Vincent.Botbol@etu.upmc.fr

ABSTRACT

In this paper, after relating a short history of the mostly unhappy relationship between static typing and JavaScript (JS), we explain a new attempt at conciliating them which is more respectful of both worlds than other approaches. As an example, we present Onyo [1], an advanced binding of the Enyo JS library for the OCaml language. Onyo exploits the expressiveness of OCaml's type system to properly encode the structure of the library, preserving its design while statically checking that it is used correctly, and without introducing runtime overhead.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Interoperability; D.3.3 [**Programming languages**]: Language Constructs and Features; D.3.4 [**Programming languages**]: Processors

General Terms

Design, Languages, Experimentation

Keywords

JavaScript, OCaml, Static Typing, Interoperability

1. JS IS NOT JAVA

The most common method to bring static typing to JS libraries is to mold them into a Java-like shape. In the early days of modern Web, several libraries emerged that simulated Java's OO operations. The goal was for mainstream developers to feel more at home in the browser, and to reuse existing development methods for Java.

We do not argue on the motivations behind this approach. Collaborative development of any non trivial application or library cannot be done without agreeing on coding rules and conventions beforehand. Even for lonesome hackers, prototypes are so flexible that many prefer the use a simple set of higher level operations to define or alter object types. High level object model libraries are thus unarguably a good idea. The point on which we disagree with many libraries is the choice of Java's object model. In the context of Java client-server applications, it is a viable option to use the language to enforce consistency throughout the entire application and understanding among developers. This for instance the valid choice made by Google in GWT [2].

But for brand new JS developments, this approach has no objective ground, and in practice, it has not convinced the designers of major libraries. A simple look at jQuery [3] or Dojo [4] shows that their implicit object model is not Java's. For data polymorphism, JS library designers simply rely on duck typing[1] instead of using a hierarchy of classes and interfaces. JS programmers tend to use a mix of functional and imperative styles, using objects as extensible records more than as encapsulated components. Even when objects are used as autonomous components, encapsulation and visibility, which are central topics in Java, are not considered important. For instance, introspection and patching of user code by libraries and vice versa are common practices.

This dogmatic choice of Java-like constructs has had a bad influence on recent Web languages and frameworks, at the expense of both prototypes and static typing. Recent typed derivatives of JS (eg. Dart [5], TypeScript [6]) are a striking example. All of them define a Java-like object model and type system. But to restore the compatibility with native JS code, their designers have introduced relaxed typing rules, making these object models and type systems more or less optional. Even the now defunct JS 2 and the sixth release of ECMAScript introduce Java style classes, validating the belief that prototypes are not a satisfactory object model.

In this paper, we argue that the expressiveness of prototypes can lead to interesting, innovative designs and should not be thrown away. For this, we show that with a little work, this expressiveness can be tamed by an appropriate static type system. For this, we present a series of experiments we made around the Ocsigen [7, 9, 10] project to handle external JS libraries. Ocsigen is based on the OCaml [11] language, which has many similarities with JS: both

*Work partially supported by the French national research agency (ANR), PWD project, grant ANR-09-EMER-009-01 and performed at the IRILL center for Free Software Research and Innovation in Paris, France.

WWW 2013 Companion, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2038-2/13/05.

[1]If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

imperative and functional with a structural object layer. However, OCaml is very different from JS on one central point: it has a very strong and static typing policy. By taking advantage of the similarities between the languages, and because OCaml's type system is more flexible and expressive than Java's, we are able to build bindings to existing libraries that manage to (1) give a faithful view of the library's architecture, (2) provide a statically typed interface and (3) do not seem too much foreign for the OCaml programmer. In the end, OCaml programs manage to be typesafe not only during pure OCaml execution but also when talking to JS libraries.

In section 2 we explain how we designed a direct mapping between OCaml's static type system for objects to JS's duck typing. In section 3, we explained how we designed static interfaces between libraries written in JS and OCaml program using them using Enyo [8] as an example.

2. TYPING JS OBJECTS

Several new languages and research works have attempted to provide a static type system for JS. Although interesting, this is not the subject of our work. We do not want to typecheck existing libraries, we accept that they have been guaranteed correct, by testing if not by typechecking. What we want to type is our usage of their APIs, and a first idea is to give static types to the data we pass to and receive from the library.

Among other paradigms, OCaml can deal with objects. In OCaml, as in JS, objects can be built on the fly and do not need to belong to any class. To reflect this in the type system, instead of using a hierarchy of names, the type of an object is simply the set of its public methods, along with the types of their arguments and return values. In other words, the type of an object is a description of its (public) structure, which can be seen as a static version of duck typing. Similarity, compatibility between types is not a matter of inheritance (nominal subtyping) but of structure comparison (structural subtyping). Thence, in order to describe the structure of JS values, we naturally started by trying to map every JS object to an equivalent OCaml object. JS methods were reflected by OCaml methods and JS attributes by explicit OCaml getters and / or setters. For this, we preprocessed OCaml's foreign function declarations to insert low level conversions between OCaml and JS objects.

For an example JS function defined as follows:

```
function makeCoord(x, y){ return { x: x, y: y } }
```

An accurate OCaml binding would be the following:

```
external make_coord
   : 'a -> 'b ->
     < x : <get : 'a ; set : 'a -> unit > ;
       y : <get : 'b ; set : 'b -> unit > >
   = "makeCoord"
```

Reading: for some argument of some generic type `'a` and some other argument of some other type `'b` (arguments and return values are separated by `->` in OCaml), returns an object with a read / write property `x` of type `'a` and a read / write property `x` of type `'b`. This may seem a bit cryptic for the unaccustomed reader but actually is trivial to a programmer familiar with OCaml objects. The types may also appear long to write, but this is not a problem since OCaml uses type inference. As in untyped languages, the programmer never writes the types of variables or functions, the compiler is clever enough to synthesize them.

With this technique, the type system ensures the OCaml programmer that his use of JS object respects their structure. Of course, this assumes that the binding correctly describes the implementation. This method also implies that it is indeed possible to determine statically the structure of JS objects. Indeed, this may not be the case for arbitrary JS code, but is enough for a lot of JS libraries. For instance, we managed to give types to most of the browser environment (DOM elements, events, etc.) and to a significant part of Google's closure library.

During these experiments, we realized that we were not really using OO programming. Objects coming from JS were used as is, without resorting to inheritance, overriding or even crafting object from the OCaml side. Indeed, JS libraries tend to use functional style callbacks rather than method overriding to customize components. As a result, the technique used in Ocsigen today is a variation involving a feature available in OCaml's type system called phantom types: types that are more precise than the actual associated values in order to refine their specification. JS objects are not seen as OCaml objects but as abstract, foreign values at runtime, removing the cost of building OCaml objects. However they are still seen as (phantom) objects from the type system point of view, so we ensure the same type safety.

In the end, by writing familiar OCaml structural object types, we manage to give static types to JS APIs, allowing OCaml programmers to write programs which are entirely type safe, including foreign calls, at no runtime cost.

3. STATIC TYPES AS AN INTERFACE

The interfaces built with the technique presented in the previous section are undeniably type safe, however, they are also mandatorily low level. Indeed, the technique requires to exhibit the structure of foreign JS objects, which may not be so useful to the OCaml programmer.

In this section, we present a different attempt at bringing a typesafe OCaml binding to a JS library. This time, we chose to use functional style instead of OO programming. Here again, we use phantom types to precise the types of foreign values at zero runtime cost, but these phantom types are high level concepts instead implementation structure.

Enyo is the toolkit developed by HP for the WebOS platform. It can also be used to develop mobile Web apps or HTML5 based Android apps. It is mostly a GUI library that mimics the native interfaces of modern hand-held devices in the browser. It provides a hierarchy of components that are (1) wrappers around HTML5 elements, (2) simple GUI items and layouts and (3) advanced widgets (the Onyx module). Enyo is interesting for us because its way of creating and manipulating components uses a JS centric design pattern very different from traditional OO component libraries. In Enyo, object constructors are decoupled from the objects they give birth to. The programmer first builds a tree of object constructors to describe its UI. Then, he instantiates its root manually to obtain the resulting tree of GUI components. In order to retrieve the instances of specific components, the programmer has to introduce unique identifiers manually. To customize the behaviour of a component, the programmer can override one of its methods, but not directly. For this, he puts in the constructor object a JS method which will be copied in the resulting object. To extend a component with some property, the programmer can put its name and value in the constructor object. When instantiated, the result is a getter, a setter and an event triggered when the value changes. This design pattern is not completely exotic and can be found in other JS libraries, but is very different from usual OO programming.

We could have chosen to hide this separation under a heavy duty OCaml object machinery which would have mapped a single OCaml object to both the constructor and the instantiated object. But we really wanted to manipulate the JS objects directly, so that

no runtime cost would be introduced, and to give them nice types which represent the underlying concepts and make their use from OCaml statically checked.

To implement Onyo, we wanted to try different solutions. So we chose to ease this experimental step by generating the binding automatically from an abstract description of the components defined by the library using an ad hoc IDL (interface description language). Onyo is thus automatically generated from the set of all components provided by Enyo's distribution. This method has an interesting side effect: it is possible to restrict or extend this set of components at will, and regenerate the bindings accordingly. In the current version of Onyo, the generated bindings have the following form.

- Object constructors and instantiated objects belong respectively to parametric types `'a kind` and `'a obj`. The `'a` type parameter is restricted to be a component from the library, for instance `button kind` or `tooltip kind`. Here, `button` and `tooltip` are just abstract names for the OCaml type system, they do not represent the structure of values: they are the aforementioned phantom types.
- Values of type `'a kind` represent object constructors. They can be obtained only by calling specific constructor functions which take as optional arguments all the possible properties and methods defined by the library for this component, as in the following example.

```
tooltip:
  ?components:any_id kind list ->
  ?modal:bool ->
  ?floating:bool ->
  ?ontap:(tooltip obj -> gesture -> bool) ->
  (* ... *) ->
  tooltip kind
```

  Once created, these values are not mutable, which is consistent with how they are used in plain Enyo. They can be added as children of other constructors to build the hierarchy.
- Once the hierarchy is built, the programmer calls the following function.

```
instanciate : 'a kind -> unit
```

- After instantiation, the programmer can use introspection to browse the instantiated component tree as he does in JS. However, we provide a more typesafe alternative through the following function.

```
instance : 'a kind -> 'a obj
```

  By storing the constructors in variables, the programmer can retrieve the corresponding components after instantiation. The retrieved object has a precise static type since the phantom type is transmitted (for example, if the programmer asks for the instance of a `button kind`, he obtains a `button obj` and not a generic component). This is implemented by generating identifiers automatically.
- Properties and methods can be accessed by global functions which can only be applied to compatible components, as in the following example.

```
scrollToBottom : [pulldown | list] obj -> unit
```

  Only values of type `pulldown obj` or of type `list obj` can be passed (the vertical bar in the phantom type reads *or*).

In the end, we obtain a functional OCaml interface to Enyo which is almost completely typesafe (the only dynamic error that can arise is if `instance` is called before the GUI is instanciated) and has a very small overhead (only the automatic identifier generation). The result is also reasonably concise and OCaml like, as in the minimal example below (a button which displays how many times it has been clicked).

```
let cpt = ref 0 in
button ~content:"0"
       ~ontap:(fun self ->
            incr cpt ;
            setContent self (string_of_int !cpt))
```

4. CONCLUSION

We have shown, using a practical use case how it is possible to define statically typed bindings to JS libraries, even when their design is a bit exotic. We have been able to achieve this result by using an expressive type system[2] and by choosing an appropriate level of abstraction. We hope that these experiment are a step in convincing the world that static typing is not as incompatible with the Web as it can seem, and that introducing static typing does not necessarily mean throwing away innovative design possibilities. In the future, we plan to try and give nice interfaces to more libraries and use them type-safely ever after.

5. REFERENCES

[1] https://github.com/klakplok/onyo.

[2] http://code.google.com/webtoolkit/.

[3] http://www.jquery.org/.

[4] http://www.dojotoolkit.org/.

[5] http://www.dartlang.org/.

[6] http://www.typescriptlang.org/.

[7] http://www.ocsigen.org/.

[8] http://www.enyojs.org.

[9] V. Balat, P. Chambart, and G. Henry. Client-server web applications with ocsigen. In *World Wide Web Conference, developers track*, 2012.

[10] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: ocsigen, a web programming framework. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 311–316. ACM, 2009.

[11] X. Leroy. The Objective Caml system release 4.01 : Documentation and user's manual. Technical report, Inria, 2012. http://caml.inria.fr.

[2]To be fair with Java's expressiveness, with a few restrictions regarding extensibility and readability, it is possible to build an equivalent of Onyo for Java. The general idea is to encode structural typing using generated Java interfaces. For instance, the `scrollToBottom` function presented earlier could be encoded as a static method. To restrict the type of its argument, a possible encoding would define an interface per component type. The type of the argument would then be a locally defined interface, which inherits the ones of `pulldown` and `list`.