Goji: an Automated Tool for Building High Level OCaml - JavaScript Interfaces

Benjamin Canou

Laboratoire d'Informatique de Paris 6 - UMR 7606[1]
Université Pierre et Marie Curie - Sorbonne Universités, 4 place Jussieu, 75005 Paris, France
benjamin.canou@lip6.fr

Abstract

We present a tool that helps writing and maintaining OCaml bindings of JavaScript libraries (browser built-in or third party), for use in `js_of_ocaml` or `obrowser`[6, 3, 5]. Goji defines a format for writing high level descriptions of these interfaces and provides a compiler from this format to OCaml. Its main strength is its ability to produce bindings that look like native OCaml libraries, hiding as possible the underlying JavaScript implementations.

Background and Motivations

There are currently two ways to interface OCaml and JavaScript. First, it is possible to use the venerable `external` syntax, usually used for interfacing OCaml and C. In this case, most of the interoperability code is written in JavaScript. The other (recommended when possible) option is to use a small set of combinators that manipulate JavaScript objects from OCaml, much as Haskell's FFI or the recent `ocaml-ctypes`[1].

Usual FFI On one hand, with the first method, library binders have to write a lot of boilerplate JavaScript code to convert the values that travel between the two worlds. Moreover, JavaScript's calling convention being much more flexible than C's, calling JavaScript through the rudimentary `external` mechanism can be tedious. In particular, it often means writing high level OCaml wrappers manually around external calls. This is for instance the case when mapping OCaml optional arguments to JavaScript ones.

Reversed FFI On the other hand, the second method removes the need to write JavaScript code and bypasses the `external` keyword. It is also very concise. But paradoxically, JavaScript objects are more visible since they become first class OCaml values. In particular, it is up to the user to convert OCaml values to JavaScript ones before feeding them to JavaScript. Moreover, to check the use of JavaScript objects statically, `js_of_ocaml` introduces a clever trick that maps the structure of external JavaScript objects to phantom OCaml object types than encode their methods and attributes. Overloading and variadicity are handled using syntactic conventions. Optional arguments can be implemented by handling `null` and `undefined` values explicitly. In the end, the programmer does not write in OCaml but in a mix of OCaml and a pseudo typed JavaScript.

To sum up, we have on one hand a solution which can lead to high level, well isolated interfaces, but requires a lot of boring code writing, and on the other hand a solution which is clever and concise but very intrusive, making the internal structure of JavaScript libraries leak inside the OCaml program using them. We designed Goji to keep the best of these two approaches while avoiding their flaws. Goji aims at allowing programmers to write bindings quickly and concisely while providing a good level of abstraction and isolation from the underlying JavaScript implementation.

An OCaml-centric Description

In order to remain concise, we simply reused a good old idea: we defined an interface description language (IDL) and wrote a compiler that generate automatically the boring interface code from it. But in order to build high-level bindings that seem native, this IDL takes an unusual form. Most IDLs, such as TypeScript[2]'s or Mozilla's ones, define bindings in a symmetric way: an object in the implementation language becomes an object in the high-level language, a method becomes a method, etc. This is acceptable when both languages have similar data structures, as in the case of the two aforementioned examples. But in our case, given that OCaml and JavaScript are quite different, we believe that this symmetry is what makes writing good OCaml bindings difficult. Thus, in order to break it, Goji's IDL is split into two parts, one dedicated to describe the desired OCaml binding, the other to map this description to JavaScript calls.

Limitations This design choice of an asymmetric approach brings advantages but also limitations. The most notable one is the difficulty to extend libraries with OCaml parts, such as writing new widget types in OCaml in a UI library. We assume this as a price to pay, but we still provide ways to work around it whenever necessary.

The IDL For now, Goji takes the form of a rough domain specific language inside OCaml. The library binder writes the syntax tree of the interface description as an OCaml value, with the help of combinators. It does not come with a concrete syntax but could be given one in the future if it becomes necessary.

Structure description The top level part of the IDL consists in defining how the generated library will look. It includes the possibility to describe modules and sections, independently from the JavaScript structure, in order to refactor the architecture of the library and make it look like a native OCaml module. This feature can also be used to increase the consistency between library bindings and even insert cross references. Documentation can be inserted at almost every place, and will be output in the `.mli` files, and thus in the generated `ocamldoc`. Moreover, every JavaScript name can be changed in order to obtain a consistent and OCaml-like naming scheme (using underscores instead of capitals, for instance).

Elements description Inside modules, it is possible to define types and values. Types can describe concrete OCaml structures, abstract extern JavaScript objects, automatically converted JavaScript values, or a combination of the three. One can also describe external global functions or data accessors. For extern JavaScript objects,

[1]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.

there is also a way to describe method callers and attribute accessors. Method and function parameters can use OCaml's optional and labeled arguments.

Mapping to JavaScript

An important part of Goji is then dedicated to map this OCaml-centric description to the JavaScript implementation. It includes notations to describe how values are converted, how arguments are passed. It also provides predefined constructs to make descriptions more concise.

Mapping values For this, we use a declarative description that can be used to generate the converters for both directions. It is possible to use it when declaring OCaml types that are bound to extern JavaScript objects. In this case, these types will appear as concrete OCaml objects and be converted on demand whenever necessary.

It is expressive enough to map, for instance, a JavaScript value of the form `{x:1, y:2, x2:3, y2:4}` to an OCaml pair of pairs `((1, 2), (3,4))` using the following declaration.

```
Type ([], "boundaries", Intern (Tuple [
   Tuple [Value [Field (Root,"x"), Float] ;
          Value [Field (Root,"y"), Float]],
   Tuple [Value [Field (Root,"x2"), Float] ;
          Value [Field (Root,"y2"), Float]] ])) ;
```

Mapping functions Several tools are provided in order to map OCaml calls to the various calling conventions defined by JavaScript developers. First, it is possible to define several OCaml functions for one in JavaScript. This is useful when binding some JavaScript patterns that would feel weird to OCaml developers. For instance, a common pattern is the getter-setter: a single function which gets a value if no argument is passed and assigns it to its argument otherwise. It is also possible to map OCaml optional arguments to the various kind of optional argument passing in JavaScript (eg. arity overloading, undefined or null placeholders, boxed argument groups). In addition, it is also possible to use an OCaml list to call a variadic function.

For instance, let us bind a JavaScript function `f` which takes an object with fields `x` and `y` and optionally a field `ofs` with the same `x` and `y` fields. Two JavaScript calls would be `f({x:1,y:3})` and `f({x:1,y:3,ofs:{x:1,y:1}})`. We can define a binding like this:

```
Function ("f", Var "f", [
  Optional ("ofs", Tuple [
    Value [Field (Field (Arg 0, "ofs"), "x"), Float] ;
    Value [Field (Field (Arg 0, "ofs"), "y"), Float] ]) ;
  Curry ("point",  Tuple [
    Value [Field (Arg 0, "x"), Float] ;
    Value [Field (Arg 0, "y"), Float] ]) ;
], None)
```

The generated function has the following signature:
`val f : ?x:float * float -> float * float -> unit`

Advanced type mappings In order to keep the IDL concise as well as to provide high-level interfaces, Goji provides predefined constructions that correspond to patterns commonly encountered in JavaScript libraries. This is actually an important design choice of the tool. For instance, many libraries use manually chosen unique string identifiers. In most cases, binding this behavior as is would result in interfaces that do not feel like OCaml, and that are not type-safe enough. Instead, it is possible to bind an OCaml abstract type to an automatic symbol generator using the `Gen_sym` type mapping. As an example, this construct can allow a JavaScript polymorphic key $\times$ value store to be interfaced in a type safe way, producing the following signature.

```
type 'ty key
val make_key : unit -> 'a key
val get : store -> 'a key -> 'a option
val set : store -> 'a key -> 'a -> unit
```

The `make_key` has been automatically added when compiling the `Gen_sym` construct in the following corresponding IDL extract.

```
Type (["ty"],"key",Gen_sym) ;
Method ("store",{ocaml_name = "get"; js_name = "data"},
  [ Curry ("key",Value [Arg 0,Abbrv (["ty"],"key") ],
  Some (Value [Root,Nullable (Param "ty")]) ;
Method ("store",{ocaml_name = "set"; js_name = "data"},
  [ Curry ("key",Value [Arg 0,Abbrv (["ty"],"key")] ;
    Curry ("value",Value [Arg 1,Param "ty"]) ],
  None) ;
```

Customizable Code Generation

Goji's current code generator produces abstract types and functions for JavaScript method bindings. We plan to write another output module that produces OCaml objects. The structure of the IDL has been designed for that. In the same vein, several generation options are planned to modulate specific parts of the code generation. In particular, event handlers could be compiled to plain OCaml callbacks, Lwt or preemptive threads.

In this context, the recommended usage is to distribute IDL files and not the OCaml code generated from them. This way, programmers can use a library consistently with their own code style. Goji can also take several IDL files and produce a single OCaml package with a single entry point for documentation. This way, a programmer can make a choice of libraries and build a complete integrated development platform for building his app.

Conclusion and Perspectives

Goji is a binding generator that is able to produce bindings that are well structured and documented and look like native OCaml libraries. It has predefined constructs for common JavaScript patterns that enable to keep IDL files reasonably simple and concise. Using it, we have been able to write high-level bindings for a part of the browser's environment, the canvas and third party libraries Raphael (vector graphics library) and Howler (audio library).

We are currently working on bindings for jQuery (a general purpose library), Enyo[4] (a component-based UI library) and Phonegap (an abstraction layer to access mobile devices hardware). We are also working on the object oriented back-end. A longer term task is to identify JavaScript idioms that should be added as built-in constructs (as the `Gen_sym` presenter earlier). In particular, it could be interesting to study the various extension / plug-in mechanisms in order to provide ways to extend JavaScript libraries with OCaml components.

References

[1] http://github.com/ocamllabs/ocaml-ctypes.

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

[3] Benjamin Canou, Vincent Balat, and Emmanuel Chailloux. O'browser: objective caml on browsers. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canaday, September 21, 2008*, pages 69--78. ACM, 2008.

[4] Benjamin Canou, Emmanuel Chailloux, and Vincent Botbol. Static typing & javascript libraries: Towards a more considerate relationship. In *World Wide Web Conference, developers track*, 2013.

[5] Benjamin Canou, Emmanuel Chailloux, and Jérôme Vouillon. How to run your favorite language in web browsers. In *World Wide Web Conference, developers track*, 2012.

[6] Jérôme Vouillon and Vincent Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 2013.