High Level OCaml-JavaScript Interfaces with Goji

Benjamin Canou
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie

Boston (MA, USA), September 24, 2013

OCaml Workshop 2013

Current Method Vs Goji

Current method: write bindings with pa_js 1 / 2

- User code use a new predefined operator ##

```
buf ## append (Js.string "my␣text")
Js.to_bool (buf ## isEmpty ())
```

- Preprocessed to generate low level calls

```
ignore (js_call_method buf "append" [| js_of_string "my␣text" |])
bool_of_js (js_call_method buf "isEmpty" [| |])
```

- And checks aigainst encoded JavaScript structures using fake OCaml object types

```
class type buffer = object
  method isEmpty : bool Js.t js_meth ;
  method append : js_string Js.t -> unit js_meth ;
end
```

Current method: write bindings with pa_js 2 / 2

PROS

- concise both for definition and calls
- reasonnably easy to write and maintain
- static typechecking at zero overcost

CONS

- visible for both binding writers and users
- introduces non-OCaml constructs and style
- fills user code with boring conversions
- not expressive enough for modern JavaScript libraries

Goji's method: describe bindings 1 / 2

Our two main goals:

- Hide the machinery from library users
- Get rid of boilerplate code / conversions

We use a good old technique: an Interface Description Language!

Goji is a tool which:

- Takes library descriptions in a specific IDL
- Generates the boring code for you
- Generates OCamlDoc from your annotations
- Does static checks and can optionally introduce dynamic checks
- Handles OCamlFind packages and JavaScript dependencies
- Has (or will have) several back-ends (abstract types / objects, concurrency)

And everything is still fresh and can be discussed !

Goji's method: describe bindings 2/2

The Interface Description Language:

- Supports OCaml features: optional arguments, complex types, modules
- Separates the desired OCaml output from its JavaScript mapping
- Predefined (and extensible) high-level constructs for conciseness
- Built as an embedded DSL : a public AST + a combinator library

In the end this original JS code

```
var sound = new Howl({
  urls: ['sounds.mp3',
         'sounds.ogg'],
  autoplay: true,
  sprite: {
    blast: [0, 2000],
    laser: [3000, 700],
    winner: [5000, 9000]
  }
});
```

can become this OCaml code

```
let sound : Howler.sound =
  Howler.make
    ~autoplay:true
    ~sprites:
      [ "blast", (0, 2000) ;
        "laser", (3000, 700) ;
        "winner", (5000, 9000) ]
    [ "sounds.mp3" ;
      "sounds.ogg" ]
```

Details & Tutorial

Creating a binding description

Form of a (set of) binding(s):

- An (or a set of) `.ml` source files
- Linked against the `goji_lib` package
- Registering packages and modules using `Goji_registry`

For instance, we create an (initially empty) package:

```
let my_package =  register_package ~doc:"My␣very␣own␣library"
                                   ~version:"3.0-0"
                                   "mylib"
```

And fill it with compilation units (components):

```
let raphael_component =
  register_component
    ~version:"3.0" ~author:"My␣Self" ~license:Goji_license.wtfpl
    ~grabber:Goji_grab.(http_get "http://self.com/~my/mylib-3.0.js")
    ~doc:"My␣very␣own␣library"
    my_package "My_lib_main"
    [ (* binding contents *) ]
```

Describing the architecture

The top-level description describes the OCaml structure:

```
[ Structure ("Utils", Doc "My␣useful␣functions", [
    Type ( (* .. *) ) ;  Method ( (* .. *) ) ;
    Inherits ( (* .. *) ) ;
  ] ;
  Structure ("Useless", Doc "My␣useless␣functions", [
    Exception ( (* .. *) ) ;  Function ( (* .. *) ) ;
  ] ;
  Function ("version", (* .. *), Doc "My␣version") ]
```

Or using the DSL:

```
[ structure "Utils" ~doc:"My␣useful␣functions" [
    def_type (* .. *) ; def_method (* .. *) ; inherits (* .. *) ;
  ] ;
  structure "Useless" ~doc:"My␣useless␣functions" [
    def_exception (* .. *) ; def_function (* .. *) ;
  ] ;
  def_function "version" ~doc:"My␣version" (* .. *) ]
```

Mapping data types / structures

Description of reversible data mappings

- Usable for both injection and extraction
- Top-level: OCaml types (tuples, records, variants, options)
- Leaves: value types (int, array, etc.) + paths inside the JavaScript structure

Notation : `type @@ location` where `location` is

- `root` (the root of the JavaScript value)
- `field location "f"`
- `cell location 3`

For instance, to map `((A, B), (C, D))` to `{ x: A, y: B, x2: C, y2: D }`

```
def_type
  ~doc:"rectangular␣boundaries␣((left,␣top),␣(right,␣bottom))"
  "boundaries"
  (public (tuple [ (tuple [ float @@ field root "x" ;
                            float @@ field root "y" ]) ;
                   (tuple [ float @@ field root "x2" ;
                            float @@ field root "y2" ]) ])) ;
```

Mapping functions / methods

A function is described by

- Its name, its parameters and return types
- What it does : specific combinators to describe the body
- How arguments are used in the body

To map OCaml arguments to JavaScript arguments, use the location `arg n`.

```
def_function "my_fun"
  [ curry_arg "x" (int @@ arg 0) ]
  (call_function "myFun")
  void
```

The body can be more complex, for instance to introduce phantom arguments:

```
def_function "my_fun"
  [ curry_arg "x" (int @@ arg 0) ]
  (seq [ set (arg 0) Const.(string "magic") ;
         call_function "myFun" ])
  void
```

Multiple call sites can be named and targeted by `arg ~site:"cs" n`.

Non demonstrated features

You didn't see it but it's available:

- Access to global JavaScript variables
- Optional / labeled arguments
- Collections (arrays, lists, assocs)
- `gen_sym`, `gen_id`, `format` constructs to get rid of "everything is a string"
- Variant types (with a notion of reversible guards)
- High-level DSL functions (e.g. `simple_enum [ "A" ; "B" ]`)
- Automatic handling of JavaScript dependencies

Conclusion & Future Work

Conclusion

README

- Available on Github:
  - The tool : `https://github.com/klakplok/goji`
  - Some bindings : `https://github.com/klakplok/goji-bindings`
- Under the CeCILL (GPL like) license
- Examples: jQuery, Raphael, Howler, Box2D

TODO

- A comprehensive introduction / tutorial (OCamlDoc is already there)
- Event handling back-ends (on their way)
- Object oriented back-end
- More, more and more bindings !

FIXME

- More static checks (e.g. some form of typechecking)
- More dynamic checks (a real / release switch)