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
Current Method VS Goji
Current method: write bindings with pa_js

- User code use a new predefined operator `##`
  
  1: buf ## append (Js.string "my\text")
  2: Js.to_bool (buf ## isEmpty ())

- Preprocessed to generate low level calls
  
  1: ignore (js_call_method buf "append" [| js_of_string "my\text" |])
  2: bool_of_js (js_call_method buf "isEmpty" [| |])

- And checks aigainst encoded JavaScript structures using fake OCaml object types
  
  1: class type buffer = object
  2:   method isEmpty : bool Js.t js_meth ;
  3:   method append : js_string Js.t -> unit js_meth ;
  4: end
PROS
- concise both for definition and calls
- reasonably 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
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!
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 **concreteness**
- Built as an embedded DSL: a public AST + a combinator library

In the end this original JS code can become this OCaml code

JS code:

```javascript
1: var sound = new Howl({
2:   urls: ['sounds.mp3',
3:     'sounds.ogg'],
4:   autoplay: true,
5:   sprite: {
6:     blast: [0, 2000],
7:     laser: [3000, 700],
8:     winner: [5000, 9000]
9:   }
10: });
```

OCaml code:

```ocaml
1: let sound : Howler.sound =
2:   Howler.make
3:     ~autoplay:true
4:     ~sprites:
5:       [ "blast", (0, 2000) ;
6:       "laser", (3000, 700) ;
7:       "winner", (5000, 9000) ]
8:       [ "sounds.mp3" ;
9:       "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:

1: let my_package = register_package ~doc:"My very own library"
2:                                ~version:"3.0-0"
3:                                   "mylib"

And fill it with compilation units (components):

1: let raphael_component =
2:     register_component
3:     ~version:"3.0" ~author:"My Self" ~license:Goji_license.wtfpl
5:     ~doc:"My very own library"
6:     my_package "My_lib_main"
7:     [ (* binding contents *) ]
Describing the architecture

The top-level description describes the OCaml structure:

1: [ Structure ("Utils", Doc "My useful functions", [ 2: Type ( (* .. * ) ); Method ( (* .. * ) ); 3: Inherits ( (* .. * ) ); 4: ] ); 5: Structure ("Useless", Doc "My useless functions", [ 6: Exception ( (* .. * ) ); Function ( (* .. * ) ); 7: ] ); 8: Function ("version", (* .. *), Doc "My version") ]

Or using the DSL:

1: [ structure "Utils" ~doc:"My useful functions" [ 2: def_type (* .. * ); def_method (* .. * ); inherits (* .. * ); 3: ] ]; 4: structure "Useless" ~doc:"My useless functions" [ 5: def_exception (* .. * ); def_function (* .. * ); 6: ] ]; 7: def_function "version" ~doc:"My version" (* .. *) ]
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 \}

```ocaml
1:  def_type
2:    ~doc:"rectangular boundaries ((left, top), (right, bottom))"
3:   "boundaries"
4:   (public (tuple [(tuple [ float @@ field root "x" ;
5:     float @@ field root "y" ]]) ;
6:     (tuple [ float @@ field root "x2" ;
7:     float @@ field root "y2" ]))));
```
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`.

```ocaml
1: def_function "my_fun"
2: [ curry_arg "x" (int @@ arg 0) ]
3: (call_function "myFun")
4: void
```

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

```ocaml
1: def_function "my_fun"
2: [ curry_arg "x" (int @@ arg 0) ]
3: (seq [ set (arg 0) Const.(string "magic") ;
4: call_function "myFun" ])
5: void
```

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