Static Typing & JavaScript Libraries

Towards a More Considerate Relationship

Benjamin Canou, Emmanuel Chailloux, Vincent Botbol
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie

Rio de Janeiro, May 13-17, 2013

World Wide Web 2013 - Developer's Track

Outline of this talk

1. Static Typing and JavaScript (how it's done)
   - in research works
   - in most advanced mainstream solutions
   - in-between: How we do it in today in OCaml
2. The proposed approach (how we propose to do it)
   - Manifesto: ideas, goals and why it's worth it
   - Concrete technical details
3. Examples ! (how we did it)
   - Binding example: Raphael.js
     (Onyo, an advanced binding to Enyo.js in the paper)
   - Hopefully a little demo

Static Typing and JavaScript

Research works

Type systems for JavaScript (and friends)

Active research field but no universal solution

- What are the underlying data types ?
  - many use them as extensible records,
  - some simulate a Java-like class hierarchy,
  - others see only hash tables, etc.
- How to handle the many styles of JavaScript programming ?
  - optional parameters (arity, type based, JSON)
  - custom event handling mechanisms
  - functions as constructors / as methods / as both, etc.
- When is a program considered type-safe ?

Decisions to take $\Rightarrow$ biased solutions

Mainstream solutions

JavaScript overlays (eg. TypeScript, Dart)

Strong pragmatic choices:

- Simple, Java-like object model and type system
- JavaScript-like syntax and concurrency model
- Possibility to introduce types progressively in existing code

But not satisfactory enough when focusing on typing:

- Not powerful enough to handle JavaScript's expressiveness
- Library authors often don't (shouldn't) refrain from using expressive features
- So relaxed typing rules are introduced to deal with libraries

In the end, two options:

- rewrite everything (incl. libraries) to gain type safety
- use existing libraries and lose type safety

What we do in OCaml

Step 1: write client side programs in OCaml

Step 2: use OCaml's object layer to describe JavaScript values

- OCaml object layer is based on structural subtyping, not nominal subtyping
  - one can define an object like this:

```
1: object
2:   val mutable st = 0
3:   method incr v = st <- st + v
4:   method get () = st
5: end
```

  - type inferred by the compiler: the set of methods and their types

```
1: < incr : int -> unit ;
2:   get : unit -> int >
```

- Much as a static interpretation of duck typing !

Step 3: describe the structure of objects coming from libraries precisely

The Approach : Typed Interfaces

Let's simplify the problem

Type the interface, not the code:

- Libraries are field proven, no need to re-check them by typing
- Let's write user code directly in a typed language
- Only ensure that libraries are used in the expected way

Materialize concepts as abstract types, don't expose the structure:

- We do not want to know how libraries represent their data
- Foreign concepts (ex. signal, circle, sound) are mapped to abstract types
- Treatments are typed according to their documentation / JavaScript code

A solution more respectful

- of the library: no need to rewrite / tweak it to use it safely
- of the language: no introduction of foreign structures

A framework for generating typed interfaces

The framework is made of:

- An interface description language (IDL)
- A compiler to produce OCaml bindings from interface descriptions
- A tool to build interface description drafts from the code / doc

A very specific IDL:

- Describes how the library will look from the typed language
- Describes how it maps to JavaScript calls using predefined constructs
- Based on idioms identified in existing JavaScript code

Application of the method

Binding Raphael.js

A typical example:

- Specialized, well delimited scope (vector graphics), portable, robust
- Fairly simple interface, reasonably documented
- Yet featuring some non trivial to type features

Practical problem: polymorphic $(\text{key} \times \text{value})$ store

- A way to store and retrieve generic data in nodes

```
1 : Element.prototype.data
2 :     = function (key, obj) {
3 :         if (arguments.length > 1)
4 :           this.d.key = obj
5 :         else
6 :           return this.d.key
7 :       }
8 : E.prototype.removeData
9 :     = function (key) {delete this.d.key}
```

- Hard to write in most typed languages (heterogeneous collections)
- Trivial to write in JavaScript, but can we type the interface ?

To obtain a high level of type safety:

- We give keys an abstract type `key`

⇒ we materialize the concept (not just strings), can document it, etc.

- We give a type parameter `t key` and link it to the data

⇒ ensures that one key is always associated to one type

- The construtor uses the IDL idiom / keyword `gen_sym`

⇒ keys are unforgeable so no type collisions

Definition in the IDL:

```
1: type t key
2:   = gen_sym
3: get (this : element, k : t key) : nullable t
4:   = method Element.data (k)
5: set (this : element, k : t key, v : t)
6:   = method Element.data (k, v)
7: remove (this : element, k : t key)
8:   = method Element.removeData (k)
```

The generated interface:

```
1: module Data : sig
2:   type 'a key
3:   val make_key : unit -> 'a key
4:   val get element ->  'a key -> 'a option
5:   val set element ->  'a key -> 'a -> unit
6:   val remove ->  'a key -> unit
7: end
```

An example use:

```
1: let color = Data.make_key () in
2: (* allocates a new 'a key                          *)
3: Data.set elt color "blue" ;
4: (* when first used, the type parameter is fixed    *)
5: (* color passes from ['a key] to [string key]     *)
6: Data.set elt color 25
7: (* will produce an error at compile time:         *)
8: (* types [int key] and [string key]  incompatible *)
```

About JavaScript and documentation

Main problems:

- Everyone (re)invents the wheel
- The missing legend symptom (conventions used but never defined)
- Higher order (functions, objects) descriptions often missing
- Examples are good, but not enough

A little twist

- A major slow-down when building typed interfaces is the lack of documentation
- But once made, typed interfaces can constitute a unified documentation

Demo

- A glimpse at the interface definition
- The generated documentation
- An OCaml app mixing several JavaScript libraries

Conclusion

In a few words: types as an added value, not a constraint.

Type safe use of JavaScript libraries is possible

- Without hurting anyone's feelings
- When helped with an automation tool
- With some work to identify the original concepts
- Can help with documenting libraries

We are building a tool

- To use JavaScript libraries from OCaml (adaptable to Scala, Haskell, etc.)
- Capable of integrating several libraries in one development platform
- As open source of course, expect news on ocsigen.org

A more complex library: binding Enyo.js

Difficult to type traits:

- Constructors and instances of components are decoupled
- Manual ID based component retrieval
- Remote event handling with custom events

Solutions:

- Automatic ID generation
- Automatic typed link between constructor and instance

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

- Abstract type for typed signal with `gen_sym`

```
1: val make_signal : unit -> 'a signal
2: val trigger : 'a signal -> 'a -> unit
3: val handle : 'a signal -> ('a -> unit) -> unit
```