ocsigen

Programmation du Client en OCaml

Benjamin Canou

Journée Francophone de Leçons sur Ocsigen

ocsigen

Préparatifs

Rappels de JavaScript
Manipulations du document

L'inévitable JavaScript

On veut faire des applications Web :

- Plus seulement des sites Web : le client devient intelligent.
- On veut donc programmer le navigateur :
  - Interagir avec l'utilisateur.
  - S'adapter au matériel (ex. capacités spécifiques aux mobiles).
  - Communiquer avec le serveur.
  - Modifier dynamiquement la page Web.

Donc on doit utiliser JavaScript :

- Le langage de script inclus dans les navigateurs grand public.
- La seule façon de programmer certains navigateurs (iOS, Windows 8, etc.)
- Plus simple pour accéder au document que les greffons.

Et on veut l'utiliser pour :

- Une riche galaxie de bibliothèques dont on veut profiter.
- Ses performances passées de ridicules à plutôt bonnes.

Fonctionnement des navigateurs

Modèle d'exécution : la boucle d'évènements

- Pas de mise à jour de l'affichage pendant la gestion d'évènements.
- Impossible de toucher à la queue d'évènements de l'instant courant.
- Exécution bloquée durant la gestion d'évènements.
- (Don't) try `javascript:setTimeout(function(){while(true){}})`.

Gestion d'évènements :

- Prendre en charge un évènement = lui affecter une fermeture JavaScript
- Évènements de ressources : matériel, requêtes réseau, etc.
- Évènements d'interface : souris, clavier, etc.
- Mécanisme particulier d'inhibition/propagation d'évènements (le bullage).
- Pour simuler un évènement : setTimeout (pour le tour prochain).

Le DOM (Document Object Model)

Accès à la représentation interne du document dans le navigateur.

- Arbre (sans cycle ni partage).
- Initialement :
  - Nœud HTML ⇝ un nœud DOM = un objet JavaScript.
  - Attribut HTML ⇝ attribut DOM = propriété JavaScript.
- Possibilité de modifier les nœuds, d'en ajouter, retirer, etc.

Comme en XML, les nœuds représentent :

- La structure (nœuds élément div, p, table) ;
- le contenu (nœuds texte) ;
- les méta-données (nœuds élément style, script, meta, etc.)

Les attributs stockent les données annexes associées aux nœuds.

- Les gestionnaires d'évènements deviennent des fonctions JavaScript.
- Les styles deviennent des objets JavaScript complexes.
- Certains attributs subissent un traitement particulier (id, class, src, etc.)

Le DOM (Document Object Model)

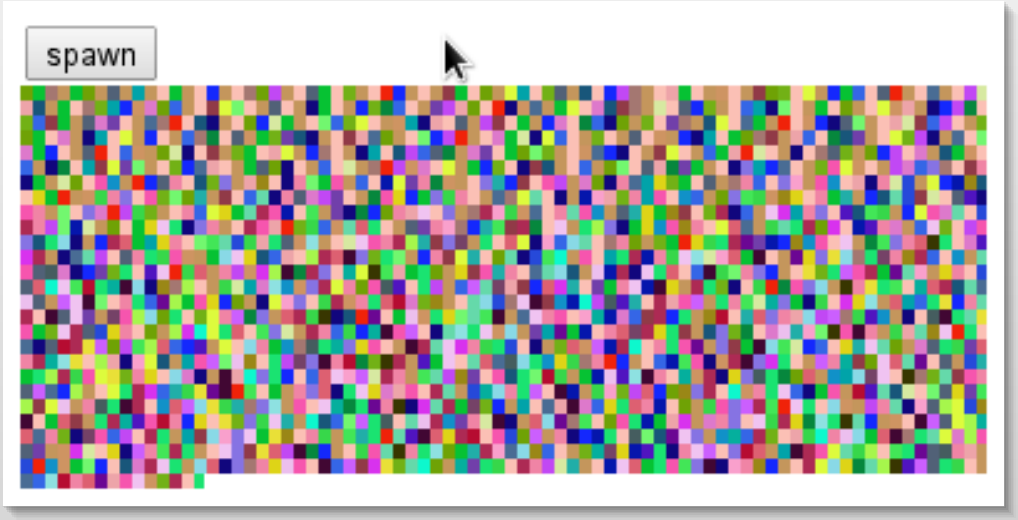
- Un petit exemple en HTML :

```
1 : <body>
2 :   <p style="font-family: 'Comic Sans MS' ">
3 :     Hello World
4 :   </p>
5 : </body>
```

- En version DOM :

```
1 : var p = document.createElement ("p");
2 : var t = document.createTextNode ("Hello World");
3 : p.style.fontFamily = "Comic Sans MS" ;
4 : p.appendChild (t);
5 : document.body.appendChild (p);
```

Une petite page interactive



D'abord une version en JavaScript, on la traduira en OCaml ensuite.

- Étape 1: clic → une case de couleur aléatoire
- Étape 2 : clic → un processus générateur de cases
- Étape 3 : clic sur une case → disparition de la couleur et mort du processus

Une petite page interactive


```
1 : <script>
2 :   function spawn () {
3 :     /* chosse random color and delay */
4 :     var timeout = Math.floor (Math.random() * 900) + 100
5 :     var r = Math.floor (Math.random() * 255)
6 :     var g = Math.floor (Math.random() * 255)
7 :     var b = Math.floor (Math.random() * 255)
8 :     /* retrieve marked HTML element */
9 :     var basket = document.getElementById ("basket")
10 :    /* create a color block */
11 :    var li = document.createElement ("span")
12 :    li.innerHTML = "    ␣"
13 :    li.style.backgroundColor = "rgb(" + r + ",␣" + g + ",␣" + b + ")"
14 :    /* insert our new item */
15 :    basket.appendChild (li)
16 :  }
17 : </script>
18 : <body>
19 :   <button onclick="spawn␣()">spawn</button><br/>
20 :   <span id="basket" style="font-size:5px"></span>
21 : </body>
```

Une petite page interactive


```
1 : <script>
2 :   function spawn () {
3 :     var timeout, r, g, b /* ... */
4 :     /* async loop */
5 :     function loop () {
6 :       var basket, li = /* ... with local r, g, b */
7 :       basket.appendChild (li)
8 :       /* delayed recursion */
9 :       setTimeout (loop, timeout)
10 :    }
11 :    loop ()
12 :  }
13 : </script>
```

Une petite page interactive


```
1 : <script>
2 :   function spawn () {
3 :     var timeout, r, g, b = /* ... */
4 :     /* store blocks */
5 :     var all = []
6 :     /* async loop */
7 :     var stop = false
8 :     function loop () {
9 :       if (stop) return;
10 :      var basket, li = /* ... */
11 :      li.onclick = function () {
12 :        /* clearTimeout (loop) */
13 :        stop = true;
14 :        for (i in all) basket.removeChild (all[i])
15 :      }
16 :      all.push (li)
17 :      basket.appendChild (li)
18 :      setTimeout (loop, timeout)
19 :    }
20 :    loop ()
21 :  }
22 : </script>
```

ocsigen

Inter-opérabilité JavaScript / OCaml

Principe
Constructions syntaxiques spécifiques
Compilation & Typage

Journée Francophone de Leçons sur Ocsigen

Comment parler à JavaScript ?

Deux possibilités pour inter-opérer avec l'environnement :

- La construction external classique
  JavaScript manipule les valeurs OCaml
- La version inversée : module Js dans Js_of_OCaml
  OCaml manipule les valeurs JavaScript

Principe de fonctionnement :

- Un type abstrait OCaml any pour les valeurs JavaScript
- Une fonction external par primitive JavaScript

```
1 : val get : any -> string -> any
2 : val set : any -> string -> any -> unit
3 : val eval : string -> any
4 : val call_method : any -> string -> any array -> any
5 :     ...
```

- L'utilisateur combine ces fonctions plutôt que d'écrire du JavaScript

Opérations de haut niveau

Pour améliorer la sûreté d'exécution, on relie :

- La structure présumée des objets JavaScript
- Des types d'objets OCaml

Sans passer par la couche objet OCaml : types fantômes.

- Les objets restent au niveau des types
- Deux modèles objets différents
- Pas de sur-coût introduit par le typage

Opérations de haut niveau

On implante ce modèle objet avec une extension de syntaxe.

- Accès au champ f de l'objet o : `o ## f`
- Affectation du champ f de l'objet o : `o ## f <- v`
- Appel de la méthode m de l'objet o : `o ## m (param1, param2, ...)`
- Construction d'un objet : `jsnew constr (param1, param2, ...)`

L'extension :

- Insère des contraintes de types
- Génère des appels aux primitives externes non typées
- Permet de simuler la surcharge
  (en supprimant tout après le dernier _ des noms de méthodes)

Le compilateur sait ensuite repérer et optimiser ces appels

Opérations de haut niveau

Le petit exemple JavaScript :

```
1 : var p = document.createElement ("p");
2 : var t = document.createTextNode ("Hello World");
3 : p.style.fontFamily = "Comic Sans MS" ;
4 : p.appendChild (t);
5 : document.body.appendChild (p);
```

Devient :

```
1 : let p = document ## createElement (Js.string "p");
2 : let t = document ## createTextNode (Js.string "Hello World");
3 : p ## style ## fontFamily <- Js.string "Comic Sans MS" ;
4 : p ## appendChild (t);
5 : document ## body ## appendChild (p)
```

Où les types sont maintenant vérifiés statiquement.

Compilation

On compile en utilisant :

- La bibliothèque standard non modifiée
- Une bibliothèque d'exécution supplémentaire
- L'extension de syntaxe

Exemple de Makefile avec ocamlfind :

```
1 : all: ex.js
2 :
3 : %.js: %.byte
4 :         js_of_ocaml $< -o $@
5 :
6 : %.byte: %.ml lwt_js_events.cmo
7 :         ocamlfind ocamlc \
8 :           -package js_of_ocaml \
9 :           -package js_of_ocaml.syntax \
10 :          -package lwt.syntax \
11 :          -syntax camlp4o \
12 :          -linkpkg -custom \
13 :           $< -o $@
```

Définition de types objets JavaScript

Le module Js définit :

- Le type paramétré +'a t des objets javascript
- Des types pour décrire les champs et méthodes +'a meth et +'a gen_prop
- Le type paramétré +'a constr pour décrire un constructeur d'objet
- Des types objets prédéfinis décrivant les objets de base JavaScript
- Des constructeurs / convertisseurs

Par exemple pour les chaînes :

```
1 : class type js_string = object
2 :   method charAt : int -> js_string t meth
3 :   method search : regExp t -> int meth
4 :   method slice : int -> int -> js_string t meth
5 :   method length : int readonly_prop
6 :   ...
7 : end
8 : val string : string -> js_string t
9 : val to_string : js_string t -> string
```

Typage

- Accès :

$$\frac{\texttt{obj : <m : <get : u> gen\_prop> Js.t}}{\texttt{obj\#\#m : u}}$$

- Affectation :

$$\frac{\texttt{obj : <m : <set : u -> unit> gen\_prop> Js.t} \quad \texttt{e : u}}{\texttt{obj\#\#m <- e : unit}}$$

- Appel de méthode :

$$\frac{\texttt{obj : <m : t\_1 -> ... -> t\_n -> u meth; ..> Js.t} \quad \texttt{e\_i : t\_i} \quad \texttt{(1 <= i <= n)}}{\texttt{obj\#\#m(e\_1, ..., e\_n) : u}}$$

- Construction :

$$\frac{\texttt{constr : (t\_1 -> ... -> t\_n -> u Js.t) Js.constr} \quad \texttt{e\_i : t\_i} \quad \texttt{(1 <= i <= n)}}{\texttt{jsnew constr (e1, ..., en) : u}}$$

Retour à OCaml depuis JavaScript

Pour définir une méthode JavaScript à partir d'une fonction OCaml :

- Un type `(-'a,+'b)meth_callback`
- Deux convertisseurs (fonction ou méthode)
- Pour un appel de méthode, la fonction OCaml prend le receveur en argument

```
1 : type (-'a, +'b) meth_callback
2 : wrap_callback (fun x y -> x + y) :
3 :   ('a -> 'b) -> (unit, 'a -> 'b) meth_callback
4 : val wrap_meth_callback :
5 :   ('a -> 'b -> 'c) -> ('a, 'b -> 'c) meth_callback
```

Pour les évènements DOM :

- Un type spécifique Dom.event_listener et son convertisseurs Dom.handler
- Une interface spécifique pour Lwt

Le même exemple en Js_of_OCaml

Étape 1 : clic → une case de couleur aléatoire

```
1 : let example () =
2 :   Dom_html.window ## onload <- Dom.handler (fun _ ->
3 :     let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :     button ## onclick <- Dom.handler (fun _ ->
5 :       let r, g, b = Random.(int 255, int 255, int 255) in
6 :       let block = color_block r g b in
7 :       let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
8 :       Js._true) ;
9 :     Js._true)
```

Avec quelques utilitaires :

```
1 : let unopt x = Js.Opt.get x (fun () -> raise Not_found)
2 : let retrieve id = unopt (Dom_html.document ## getElementById (Js.string id))
3 : let color_block r g b =
4 :   let block = Dom_html.createSpan Dom_html.document in
5 :   let css_color = CSS.Color.string_of_t (CSS.Color.rgb r g b) in
6 :   let _ = block ## style ## backgroundColor <- Js.string css_color in
7 :   let spaces = Dom_html.document ## createTextNode (Js.string "    ") in
8 :   let _ = block ## appendChild ((spaces :> Dom.node Js.t)) in
9 :   block
```

Le même exemple en Js_of_OCaml

Étape 2 : clic → un processus générateur de cases

```
1 : let example () =
2 :   Dom_html.window ## onload <- Dom.handler (fun _ ->
3 :     let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :     button ## onclick <- Dom.handler (fun _ ->
5 :       let r, g, b, d = Random.(int 255, int 255, int 255, 200. +. float 800.) in
6 :       let rec loop () =
7 :         let block = color_block r g b in
8 :         let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
9 :         let _ = Dom_html.window ## setTimeout (Js.wrap_callback loop, d) in
10 :        ()
11 :      in
12 :      loop () ;
13 :      Js._true) ;
14 :    Js._true)
```

Le même exemple en Js_of_OCaml

Étape 3 : clic sur une case → disparition de la couleur et mort du processus

```
1 : let example () =
2 :   Dom_html.window ## onload <- Dom.handler (fun _ ->
3 :     let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :     button ## onclick <- Dom.handler (fun _ ->
5 :       let r, g, b, d = Random.(int 255, int 255, int 255, 200. +. float 800.) in
6 :       let stop = ref false and all = ref [] in
7 :       let rec loop () =
8 :         if not !stop then (
9 :           let block = color_block r g b in
10 :          let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
11 :          let _ = Dom_html.window ## setTimeout (Js.wrap_callback loop, d) in
12 :          let _ = block ## onclick <- Dom.handler (fun _ ->
13 :            List.iter
14 :              (fun o -> ignore (basket ## removeChild ((o :> Dom.node Js.t))))
15 :              !all ;
16 :            stop := true ;
17 :            Js._true) in
18 :          all := block :: !all)
19 :       in
20 :       loop () ; Js._true) ;
21 :     Js._true)
```

Le même exemple en Js_of_OCaml + Lwt_js_events 1 / 4

Étape 1 : clic → une case de couleur aléatoire

```
1 : let example_lwt () =
2 :   lwt _ = wait_for_page_load () in
3 :   let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :   let rec loop () =
5 :     lwt _ = Lwt_js_events.click button in
6 :     let r, g, b, d = Random.(int 255, int 255, int 255, 0.2 +. float 0.8) in
7 :     let block = color_block r g b in
8 :     let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
9 :     loop ()
10 :  in
11 :  loop ()
12 :
13 : let _ = example_lwt ()
```

Pour transformer l'évènement window.onload en promesse :

```
1 : let wait_for_page_load () =
2 :   let t, u = Lwt.wait () in
3 :   let _ = Dom_html.window ## onload <-
4 :             Dom.handler (fun _ -> Lwt.wakeup u () ; Js._true) in
5 :   t
```

Le même exemple en Js_of_OCaml + Lwt_js_events 2 / 4

Autre option : utiliser une boucle prédéfinie :

```
1 : let example_lwt () =
2 :   wait_for_page_load () >>= (fun () ->
3 :     let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :     Lwt_js_events.buffered_loop Lwt_js_events.click button
5 :       (fun evt _ ->
6 :         let r, g, b, d = Random.(int 255, int 255, int 255, 0.2 +. float 0.8) in
7 :         let block = color_block r g b in
8 :         let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
9 :         return ()))
```

Le même exemple en Js_of_OCaml + Lwt_js_events 3 / 4

Étape 2 : clic → un processus générateur de cases

```
1 : let example_lwt () =
2 :   wait_for_page_load () >>= (fun () ->
3 :     let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :     Lwt_js_events.async_loop Lwt_js_events.click button
5 :       (fun evt _ ->
6 :         let r, g, b, d = Random.(int 255, int 255, int 255, 0.2 +. float 0.8) in
7 :         let rec loop () =
8 :           let block = color_block r g b in
9 :           let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
10 :          Lwt_js.sleep d >>= loop
11 :        in loop ()))
```

Le même exemple en Js_of_OCaml + Lwt_js_events 4 / 4

Étape 3 : clic sur une case → disparition de la couleur et mort du processus

```
1 : let example_lwt () =
2 :   wait_for_page_load () >>= (fun () ->
3 :     let basket, button = retrieve "basket", retrieve "spawnbutton" in
4 :     Lwt_js_events.async_loop Lwt_js_events.click button
5 :       (fun evt _ ->
6 :         let r, g, b, d = Random.(int 255, int 255, int 255, 0.2 +. float 0.8) in
7 :         let continue = Lwt_switch.create () in
8 :         let rec loop () =
9 :           if Lwt_switch.is_on continue then
10 :            let block = color_block r g b in
11 :            let _ = basket ## appendChild ((block :> Dom.node Js.t)) in
12 :            Lwt_switch.add_hook (Some continue) (fun () ->
13 :              let _ = basket ## removeChild ((block :> Dom.node Js.t)) in
14 :              return ()) ;
15 :            ignore_result
16 :              (Lwt_js_events.click block >>= (fun _ ->
17 :                Lwt_switch.turn_off continue)) ;
18 :            Lwt_js.sleep d >>= loop
19 :          else
20 :            return ()
21 :        in loop ()))
```

ocsigen

Faire mieux que JavaScript

Utilisation bien typée de bibliothèques
À la recherche du bon modèle de gestion d'évènements

Journée Francophone de Leçons sur Ocsigen

Utilisation bien typée d'Enyo

Une bibliothèque :

- Complète de composants d'interface graphique pour mobiles
- Issue de WebOS

On voudrait :

- Pouvoir utiliser Enyo de façon bien typée
- Facilement ajouter des composants
- Utiliser des types OCaml et ne pas voir la sous-couche

On utilise un IDL pour décrire la structure des composants.

- Directement en OCaml
- On génère un module OCaml d'interface depuis l'IDL
- Ajouter un composant = regénérer la bibliothèque

On encode le modèle objet nominal avec des étiquettes.
(variants polymorphes en types fantômes)

Utilisation bien typée d'Enyo

Exemple de définition composant :

- Contrôle déplaçable au doigt ou à la souris Slideable
- Hérite de Control
- Définit des attributs et méthodes

```
1 : Type_gen (Control._control, "Slideable",
2 :   [ Method ("toggleMinMax", [Unit; Unit]);  Method ("animateTo", [Int;Unit]); (* ..
3 :   [ Attribute("axis", String, false);
4 :     Attribute("value", Int, false); Attribute("unit", String,false);
5 :     Attribute("min", Int,false); Attribute("max", Int,false);
6 :     Attribute("accelerated", String,false); (* ... *) ], [])
```

Code généré (l'héritage est déplié) :

```
1 : val slideable :
2 :   ?components:any_id kind list ->
3 :   ?axis:string -> ?value:int -> ?unit:string ->
4 :   ?min:int -> ?max:int ->
5 :   (* ... *)
6 :   -> ?ontap:([`SLIDEABLE] obj -> any_id obj -> [`GESTURE] obj -> bool)
7 :   -> unit -> [>`SLIDEABLE] kind
```

Utilisation bien typée d'Enyo

Modèle objet un peu farfelu :

- Le programmeur crée une description de l'interface
- Il instancie cette description pour obtenir son interface
- Il retrouve ses petits avec des id placés dans la description.

Pour typer ce fonctionnement :

- On introduit deux types `'a kind` et `'a obj`
- On génère des identifiants
- On donne une fonction `instance : 'a kind -> 'a obj`
- On donne un mécanisme de transtypage sûr (tests dynamiques)

Au final, on a :

- Donné une interface typée à la bibliothèque
- Conservé autant que possible l'expressivité du modèle

Typer jQuery

La bibliothèque que tout le monde utilise :

- Couche de simplification/compatibilité du DOM
- Simplification de la concurrence, gestion d'evènements, animations, etc.
- Sélection dans le dom avec syntaxe de sélecteurs CSS

Faire mieux qu'un simple biding : utilisation statiquement typée

- Améliorations faciles : ex. constantes globales $\rightarrow$ types somme
- PLus avancées : typer correctement les gestionnaires d'évènements
- D'ouverture : typer les sélecteurs CSS ?

Gestion d'évènements de haut niveau

Il faut gérer les évènements :

- De façon propre et pratique
- Compatible avec les bibliothèques externes
- De façon uniforme

Tentatives déjà effectuées :

- Concurrence préemptive
- Programmation réactive
- Programmation réactive fonctionnelle
- Combinateurs spécifiques

Méthode recommandée : `Lwt_js_events`

ocsigen

À TAAAAABLE !

Journée Francophone de Leçons sur Ocsigen